1995

Multidimensional wavelets and their applications

Eugene Garry Ionel
San Jose State University

Multidimensional Wavelets and their Applications

A Thesis
Presented to the Faculty of
The Department of Mathematics and
Computer Science
San Jose State University

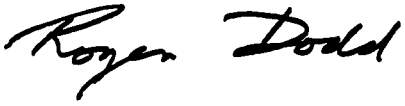
In Partial Fulfillment
of the Requirement for the Degree
Master of Science

by
Eugene Garry Ionel
May 1995

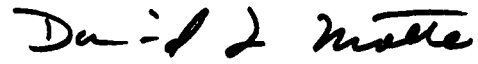
APPROVED FOR THE DEPARTMENT OF MATHEMATICS

Dr. Hedley Morris

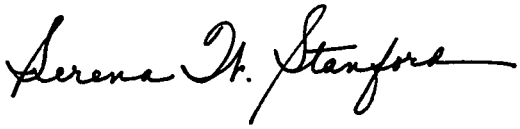
Dr. Roger Dodd



Dr. David Motte



APPROVED FOR THE UNIVERSITY



ABSTRACT

Multidimensional Wavelets and their Applications

By Eugene Garry Ionel

This thesis provides detailed description of wavelet research and application of Describe Wavelet Transform (DWT) for solving the problem of image compression. The main focus was made on Daubechies wavelets with different filters. We studied monochrome black and white images with 128 by 128 pixels. We were limited by 16MB RAM on our 486DX machine with forced us to make changes in the algorithms and methods used in our program. The results of reconstruction of compressed images were quite unexpected. We found that with truncation of the data quality of reconstructed image is not diminishing uniformly in different areas of image. We found that boundaries of objects are very stable. You can easy recognize them on the reconstructed image with compression ratio 1:100. We discovered that the most unstable areas are regions with extreme colors: absolutely black and absolutely white. They can change color from one to another with little changes in compression ratio. The multiresolution analysis was used as a practical tool for developing the programming application. The data extension problem was examined.

More information is available from the author by calling at (408)448-1674 or by e-mail: ionel@netcom.com

ACKNOWLEGMENTS

I wish to express my appreciation to Professor Hedley Morris for assistance, guidance, and encouragement given in the course of writing this thesis.

I also wish to thanks to Professor Roger Dodd and Professor David Motte for being on Comprehensive Oral Examination Committee.

Eugene Garry Ionel

Contents

List of Figures

Chapter 1

Overview

1.1 Origins (pre-history)

The word wavelet itself probably comes from the French *ondelettes*. The French probably got *ondelettes* from American seismologists who used the word wavelet.

Wavelet theory involves representing general functions in terms of simpler, fixed building blocks at different scales and positions. This has been found to be a useful approach in several different areas. For example, we have subband filtering techniques, quadrature mirror filters, pyramid schemes, etc., in signal and image processing, while in mathematical physics similar ideas are studied as a part of the theory of Coherent States. Wavelet theory represents a useful synthesis of these different approaches.

In abstract mathematics, it has been known for quite some time that techniques based on Fourier transformations are not quite adequate for many

problems and so-called *Littlewood-Paley techniques* often are effective substitutes. These techniques were initially developed in the 30's to understand, among other things, summability properties of Fourier series and boundary behavior of analytic functions. In the 50's and 60's, these developed into powerful tools for studying other things, such as solutions of partial differential equations and integral equations. It was realized that they fit into *Calderon-Zygmund theory,* an area of harmonic analysis that is still very heavily researched.

In the early 80's, Stromberg [105] discovered the first orthogonal wavelets. This was done in the context of trying to further understand Hardy spaces, as well as other spaces used to measure the size and smoothness of functions. A discrete version of the Calderon formula had also been used for similar purposes and long before this there were results by Haar [103], Franklin [41] and others.

Independent from these developments in harmonic analysis, Alex Grossmann, Jean Morlet, and their coworkers studied the wavelet transform in its continuous form [50], [51], [52]. The theory of "frames" [31] provided a suitable general framework for these investigations.

In the early to mid 80's, several groups, perhaps most notably the one associated with Yves Meyer and his collaborators, independently realized, with some excitement, that tools from Calderon-Zygmund theory, in particular the Littlewood-Paley representations, had discrete analogs and could give a unified view of many of the results in harmonic analysis. Also, one started to understand that these techniques could be effective substitutes

for Fourier series in numerical applications. As the emphasis shifted more towards the representations themselves, and the building blocks involved, the name of the theory also shifted. Alex Grossmann and Jean Morlet suggested the word "wavelet" for the building blocks, and what earlier had been referred to as Littlewood-Paley theory, now started to be called wavelet theory.

Pierre Gilles Lemarie and Yves Meyer, independent of Stromberg, constructed new orthogonal wavelet expansions. With the notion of multiresolution analysis, introduced by Stephane Mallat and Yves Meyer, a systematic framework for understanding these orthogonal expansions was developed [77], [78], [79]. It also provided the connection with quadrature mirror filtering. Soon, Ingrid Daubechies [30] gave a construction of wavelets, non-zero only on a finite interval and with arbitrarily high, but fixed, regularity. This takes up to a fairly recent time in the history of wavelets theory. Several people have made substantial contributions to the field over the past few years.

Still earlier roots of modern wavelet theory are to be found in the so called Gabor Transform (1943) and its generalization to the Weyl-Heisenberg Transform. Using modern language, we will start with a function g, called the analyzing wavelet (picked by the user in a special way), and a function f, the signal to be analyzed. One forms

$$F(\gamma,\tau) = \int_{-\infty}^{\infty} f(x)g(x-\gamma)e^{2\pi i\tau x}dx. \tag{1.1}$$

F is called a Weyl-Heisenberg transform. Gabor concentrated on $g(x) =$

$e^{-x^2/2}$, which is local in time and frequency. In general F overdetermines f and given F we can recover f in a continuous liner fashion. This being so, it is natural to inquire it from the discretization,

$$F(m,n) = \int_{-\infty}^{\infty} f(x)g(x - m\gamma)e^{2\pi i n\tau x}dx, \tag{1.2}$$

one can continuously recover f. If this is so, the collection of functions $g(x - m\gamma)e^{2\pi i n\tau x}$ is called a Weyl-Heisenberg frame. Weyl-Heisenberg frames are useful in quantum mechanics - the Weyl-Heisenberg transform is important in radar theory, related as it is to the radar cross ambiguity function and the Wigner-Ville distribution.

One natural question to ask is, given a frame, how efficient is it? Put another way, can one select g, γ, and τ so that the frame $\gamma(x - m\gamma)e^{2\pi i n\tau x}$ is an orthonormal basis for L^2 ?

The discouraging answer is given by the Balian-Low Theorem [108]:

1 If $\gamma\tau > 1$, no frames exist.

2 If $\gamma\tau = 1$, orthonormal bases are possible, but either g' or $(\hat{g})'$ decays "slowly".

3 If $\gamma\tau < 1$, there are frames, but none of them are orthonormal bases.

What this means is that the information given by $F(m,n)$ is redundant, or if not redundant is not well localized in phase space.

Interest in this kind of Fourier analysis flagged, despite the importance in radar theory.

More recently, a group other than the Heisenberg group has been emphasized, and an alternative analysis has surfaced. Now we concentrate on the affine group of the line, and following Grossmann and Morlet, select an analyzing function $g(x)$ (subject to $\int_{-\infty}^{\infty} g(x)dx = 0$ for technical reasons), for a function f to be analyzed:

$$F(\gamma, \tau) = \sqrt{\gamma} \int_{-\infty}^{\infty} f(x)g(\gamma x - z)dx. \tag{1.3}$$

Again we can continuously recover f from F. The discretization selected in this case is:

$$F(m, n) = 2^{m/2} \int_{-\infty}^{\infty} f(x)g(2^m x - n)dx. \tag{1.4}$$

Here the various versions of the analyzing function in the integrand, $2^{m/2}g(2^m x - n)$, are all translates of the same shape curve, dilated or expanded by a power of 2. Consequently, if g is compactly supported, this analysis of f gives great temporal resolution, and great phase space resolution at high frequencies.

1.2 Simple 1-dimensional (1-D) example (Haar orthogonal basis)

Orthogonal wavelet expansions are an attempt to improve on Fourier series and other classical expansions. To explain the kind of improvement we are after we examine a simple case. Consider a real-valued function $f(x)$ on the

interval $[0, 1]$. You can expand it in a Fourier series

$$f(x) = b_0 + \sum_{1}^{\infty}(b_k cos 2\pi kx + a_k sin 2\pi kx), \tag{1.5}$$

or you can expand it in a Haar function series

$$f(x) = b_0 + \sum_{j=0}^{\infty}\sum_{k=0}^{2^j-1} c_{jk}\varphi(2^j x - k), \tag{1.6}$$

where $\varphi(x)$ is the function defined by

$$\varphi(x) = \begin{cases} 1 & \text{if } 0 \leq x < 1/2, \\ -1 & \text{if } 1/2 \leq x < 1, \\ 0 & \text{otherwise} \end{cases} \tag{1.7}$$

The graph of the generator of the Haar functions.

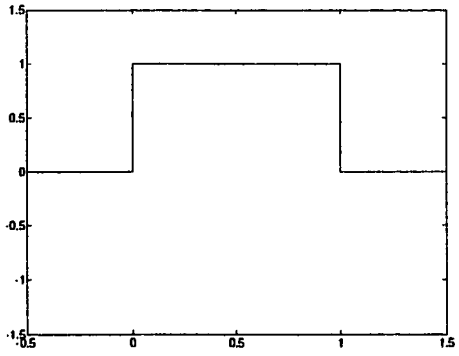


Figure 1.1

(see Figure 1.1). Both series are examples of expansions in terms of orthogo-

nal functions in $L^2(0,1)$. Thus there are simple formulas for the coefficients. The $\varphi(2^j x - k)$ are orthogonal, but not normalized. The Fourier series are not well localized in space; if you are interested in the behavior of $f(x)$ on a subinterval $[a,b]$ you need to involve all the Fourier coefficients. On the other hand, the Haar series is very well localized in that to restrict attention to the subinterval $[a,b]$ you need only take the sum in (1.7) over those indices for which the interval $I_{jk} = [2^{-j}k, 2^{-j}(k+1)]$ (the support of $\varphi(2^j x - k)$) intersects $[a,b]$. Furthermore, the partial sums of the Haar series (summing $0 \leq j \leq N$) clearly represent an approximation to f taking into account details on the order of magnitude 2^{-N} or greater. These two properties, *localization in space* and *scaling,* are the hallmarks of wavelet expansions. In addition, the Haar functions are created out of a single function φ by dyadic dilations and integer translations. Essentially the same property is shared by all the wavelet bases we will discuss, and may in fact be taken as an approximate definition of a wavelet expansion.

The wavelet expansions we are going to construct can be thought of as generalizations of the Haar series, in which the function φ is replaced by smoother cousins. Before we can say exactly what properties we want these functions to have, and how we can go about constructing them, it is useful to backtrack and see exactly how the Haar functions arise. It will turn out to be easier if we consider the whole line to be the domain of our functions.

We begin with the function φ, the characteristic function of the interval $[0,1]$. Surely this is one of the simplest functions one can imagine, but it is chosen because it has two important properties:

(i) The translates of φ by integers, $\varphi(x-k), k \in \mathbf{Z}$ form an orthonormal set of functions for $L^2(\mathbf{R})$;

(ii) φ is *self-similar.* If you cut the graph in half then each can be expanded to recover the whole graph. This property can be expressed algebraically by *the scaling identity*

$$\varphi(x) = \varphi(2x) + \varphi(2x-1). \tag{1.8}$$

We will call φ the *scaling* function. The scaling identity essentially determines φ up to a constant multiple. The significance of the scaling identity is the following: Let V_0 denote the linear span of the functions $\varphi(x-k), k \in \mathbf{Z}$ or by abuse of notation the closure in $L^2(\mathbf{R})$ of this span, $\sum_{k=-\infty}^{\infty} a_k \varphi(x-k)$ with $\sum |a_k|^2 < \infty$. This is a natural space to consider in view of (i), since the functions $\varphi(x-k)$ form an orthonormal basis for V_0. Of course V_0 is not all of L^2, it is the subspace of piecewise constant functions with jump discontinuities at $\mathbf{Z}$. We can get a large space by rescaling. Let $1/2\mathbf{Z}$ denote the lattice of half-integers $k/2, k \in \mathbf{Z}$, and let V_1 denote the subspace of L^2 of piecewise constant functions with jumps at $1/2\mathbf{Z}$. It is clear that $f(x) \in V_0$ if and only if $f(2x) \in V_1$, and the functions $2^{1/2}\varphi(2x-k)$ form an orthonormal basis for V_1 (the factor $2^{1/2}$ is thrown in to make the normalization $||2^{1/2}\varphi(2x-k)||_2 = 1$ hold). The scaling identity (1.8), or more properly its translated version

$$\varphi(x-k) = \varphi(2x-2k) + \varphi(2x-2k-1), \tag{1.9}$$

says exactly $V_0 \subseteq V_1$, since a basis for V_0 is explicitly represented as linear combinations of basis elements of V_1. Of course the containment $V_0 \subseteq V_1$ is clear from the description of the spaces V_0 and V_1 in terms of locations of jump discontinuities, but in the generalizations to come there will be no such simple discontinuities.

The whole story can now be iterated, both up and down the dyadic scale. The result is an increasing sequence of subspaces V_j for $j \in \mathbf{Z}$, where V_j consists of the piecewise-constant L^2 functions with jumps at $2^{-j}\mathbf{Z}$. The functions $2^{j/2}\varphi(2^j x - k)$ for $k \in \mathbf{Z}$ form an orthonormal basis for V_j. We can pass back and forth among the space V_j by rescaling: $f(x) \in V_j$ if and only if $f(2^{k-j}x) \in V_k$, and the scaling identity (1.8) , suitably rescaled, says $V_j \subseteq V_k$ if $j \leq k$. The sequence V_j is an example of what is called *a multiresolution analysis.* There are two other properties of V_j that are significant, namely

$$\bigcap_{j \in \mathbf{Z}} V_j = \{0\} \tag{1.10}$$

and

$$\bigcup_{j \in \mathbf{Z}} V_j \text{ is dense in } L^2. \tag{1.11}$$

In view of (1.11) it would seem tempting to try to combine all the orthonormal bases $\{2^{j/2}\varphi(2^j x - k)\}$ of V_j into one orthonormal basis for $L^2(\mathbf{R})$. But although $V_j \subset V_{j+1}$, the orthonormal basis $\{2^{j/2}\varphi(2^j x - k)\}$ for V_j is not contained in the orthonormal basis $\{2^{(j+1)/2}\varphi(2^{j+1}x - k)\}$ for V_{j+1}. (Indeed, there are distinct elements in the two orthonormal bases that are not orthogonal to each other.) So our first naive attempt to obtain an orthonormal

basis for $L^2(\mathbf{R})$ is flawed. Can we fix it up ?

Let's go back. Since $V_0 \subseteq V_1$ and we have orthonormal basis for V_0 of the form $\{\varphi(x-k)\}$, so we will try to complete an orthonormal basis of V_1 by adjoining functions of the form $\{\psi(x-k)\}$ for some function ψ? This is the same thing as asking for an orthonormal basis of the desired form for the orthogonal complement of V_0 in V_1, which we denote W_0, so $V_1 = V_0 \oplus W_j$ (Hilbert space direct sum).

The answer is easy. We want to take ψ exactly to be the Haar function generator defined as before. Note that ψ can be expressed in terms of φ by

$$\psi(x) = \varphi(2x) - \varphi(2x-1), \tag{1.12}$$

which is very reminiscent of the scaling identity. So $\{\psi(x-k)\}$ forms an orthonormal basis for W_0. But now we can rescale the space W_0, so

$$V_{j+1} = V_j \oplus W_j \tag{1.13}$$

and $\{2^{j/2}\psi(2^j x - k)\}_k \in \mathbf{Z}$ is an orthonormal basis for W_j. If we combine conditions (1.10), (1.11), (1.13) we obtain

$$L^2(\mathbf{R}) = \bigoplus_{j=-\infty}^{\infty} W_j. \tag{1.14}$$

Since the spaces ψ_j are all mutually orthogonal, we can now refine our naive attempt and combine all the orthogonal bases for W_j into one grand orthonormal basis $\{2^{j/2}\psi(2^j x - k)\}_{j\in\mathbf{Z},k\in\mathbf{Z}}$ The only change is that we have

replaced the scaling function φ by the wavelet ψ. This gives the Haar series basis for the whole line. There is a minor variation on this theme that is perhaps more closely related to the Haar expansion on the unit interval: instead of (1.14), we can also write

$$L^2(\mathbf{R}) = V_0 \oplus \{\bigoplus_{j=0}^{\infty} W_j\} \tag{1.15}$$

and then combine the basis $\{\varphi(x-k)\}_{k\in\mathbf{Z}}$ for V_0 with the bases $\{2^{j/2}\psi(2^{1/2}x-k)\}$ for ψ_j, with $j \leq 0$ to obtain an orthonormal basis for $L^2(\mathbf{R})$.

1.3 Multiresolution analysis.

We start by defining multiresolution analysis of $L^2(\mathbf{R})$ is defined as a sequence of closed subspaces V_j of $L^2(\mathbf{R}), j \in \mathbf{Z}$, with the following properties [79]:

1. $V_j \subseteq V_{j+1}$,
2. $v(x) \in V_j \iff v(2x) \in V_{j+1}$,
3. $v(x) \in V_0 \iff v(x+1) \in V_0$,
4. $\bigcup_{j=-\infty}^{\infty} V_j$ is dense in $L^2(\mathbf{R})$ and $\bigcap_{j=-\infty}^{\infty} V_j = \{0\}$,
5. A *scaling function* $\varphi \in V_0$, with a non-vanishing integral, exists such that the collection $\{\varphi(x-l)_{l\in\mathbf{Z}}\}$ is a basis of V_0.

Let us make a couple of simple observations concerning this definition. Since $\varphi \in V_0 \subseteq V_1$, a sequence $(h_k) \in L^2(\mathbf{Z})$ exists such that the scaling function satisfies

$$\varphi(x) = 2\sum_k h_k \varphi(2x - k). \tag{1.16}$$

We have a dilation equation. By integrating both sides of (1.16), and dividing by the (non-vanishing) integral of φ, we see that

$$\sum_k h_k = 1 \tag{1.17}$$

If the scaling function belongs to L^2, it is, under very general conditions, uniquely defined by its refinement equation and the normalization,

$$\int_{-\infty}^{\infty} \varphi(x)dx = 1. \tag{1.18}$$

There are fast algorithms that use the refinement equation to evaluate the scaling function φ at dyadic points ($x = 2^{-j}k, j, k \in \mathbf{Z}$) [30], [102]. In many applications, we never need the scaling function itself; instead we may often work directly with the h_k.

The spaces V_j will be used to approximate general functions. This will be done by defining appropriate projections onto these spaces. Since the union of all the V_j is dense in $L^2(\mathbf{R})$, we are guaranteed that any given function can be approximated arbitrarily close by such projections.

To be able to use the collection $\{\varphi(x - l), l \in \mathbf{Z}\}$ to approximate even the simplest functions (such as constants), it is natural to assume that the

scaling function and its integer translates form a *partition of unity,* or, in other words,

$$\forall x \in \mathbf{R} : \sum_k \varphi(x-k) = 1. \tag{1.19}$$

1.4 The Haar wavelet

We already talked about orthonormal Haar basis. The example of Haar wavelet is easier to draw than to describe (Figures 1.2a, 1.2b, 1.2c, 1.2d), [103].

$\phi(x)$

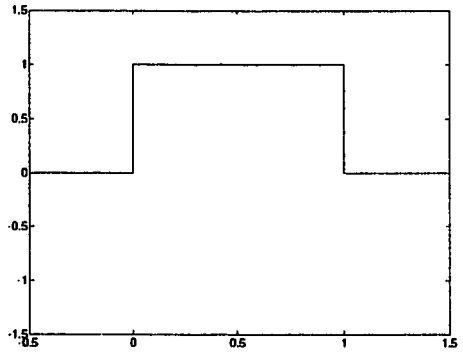


Figure 1.2a

$\psi(x)$

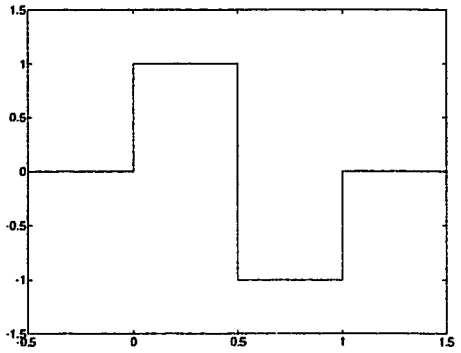


Figure 1.2b

$\psi(2x)$

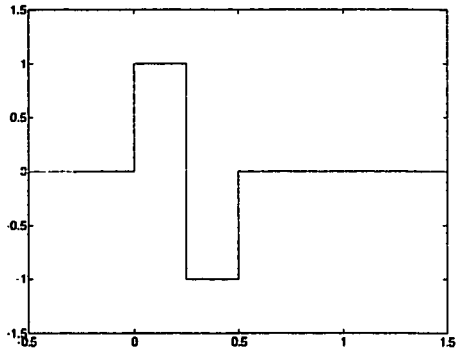


Figure 1.2c

$\psi(2x-1)$

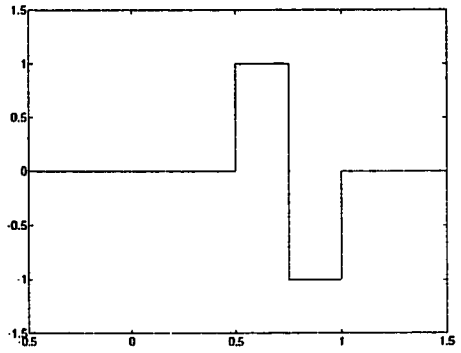


Figure 1.2d

Already we saw the two essential operations: *translation* and *dilation*. The step from $\psi(2x)$ to $\psi(2x-1)$ is translation. The step from $\psi(x)$ to $\psi(2x)$ is dilation. Starting from a single function, the graphs are shifted and compressed. The next level contains $\psi(4x), \psi(4x-1), \psi(4x-2), \psi(4x-3)$. Each is supported on an interval length $1/4$. In the end we have Haar's infinite family of functions:

$$\psi_{jk}(x) = \psi(2^j x - k) \tag{1.20}$$

(together with $\phi(x)$).

When the range of indices is $j \geq 0$ and $0 \leq k < 2^j$, these functions form a remarkable basis for $L^2[0,1]$. We extend it below to a basis for $L^2(\mathbf{R})$.

The four functions in (Figure 1.2a, 1.2b, 1.2c, 1.2d) are piecewise constant. Every function that is constant on each quarter-interval is a combina-

tion of these four. Moreover, the inner product $\int \phi(x)\psi(x)dx$ is zero - and so are the inner products. This property extends to all j and k : *The translations and dilations of W are mutually orthogonal.* We accept this as the definition of a wavelet, although variations are definitely useful in practice. The goal looks easy enough, but the example is deceptively simple.

It is reminiscent of the Walsh basis but the difference is important. For Walsh and Hadamard, the last two basis functions are changed to $\psi(2x) \pm \psi(2x-1)$. All of their "binary sinusoids" are supported on the whole interval $0 \leq x \leq 1$. This global support is the one drawback to sines and cosines; otherwise, Fourier is virtually unbeatable. To represent a local function, vanishing outside a short interval of space or time, a global basis requires extreme cancellation. Reasonable accuracy needs many terms of the Fourier series. *Wavelets give a local basis.*

So if signal $f(x)$ disappears after $x = 1/4$, only a quarter of the later basis functions are involved. The wavelet expansion directly reflects the properties of f in physical space, while the Fourier expansion is perfect in frequency space.

The great value of orthogonality is to make expansion coefficients easy to compute. Suppose the values of $f(x)$, constant on four quarter-intervals, are 9, 1, 2, 0. Its Haar wavelet expansion expresses this vector y as a

combination of the basis functions:

$$\begin{bmatrix} 9 \\ 1 \\ 2 \\ 0 \end{bmatrix} = 3 \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} + 2 \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix} + 4 \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ -1 \end{bmatrix}.$$

The wavelet coefficients b_{jk} are 3, 2, 4, 1; they form the wavelet transform of f. The connection between the vectors y and b is the matrix ψ_4, in whose orthogonal columns the graphs of (Figure 1.2a, 1.2b, 1.2c, 1.2d): $y = \psi_4 b$ is

$$\begin{bmatrix} 9 \\ 1 \\ 2 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & -1 & 0 \\ 1 & -1 & 0 & 1 \\ 1 & -1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \\ 4 \\ 1 \end{bmatrix}.$$

This is exactly comparable to the *Discrete Fourier Transform,* in which $f(x) = \sum a_k e^{ikx}$ stops after four terms. Now the vector y contains the values of f at four points: $y = F_4 a$ is

$$\begin{bmatrix} f(0) \\ f(\pi/2) \\ f(2\pi/2) \\ f(3\pi/2) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & i^2 & i^3 \\ 1 & i^2 & i^4 & i^6 \\ 1 & i^3 & i^6 & i^9 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}.$$

This Fourier matrix also has orthogonal columns. The n by n matrix F_n follows the same pattern, with $\omega = e^{2\pi i/n}$ in place of $i = e^{2\pi i/4}$. Multiplied by $1/\sqrt{n}$ to give orthonormal columns, it is the most important of all unitary

matrices. The wavelet matrix sometimes offers modest competition.

To invert a real orthogonal matrix we transpose its complex conjugate. After accounting for the factors that enter when columns are not unit vectors, the inverse matrices are

$$\psi_4^{-1} = 1/4 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 2 & -2 & 0 & 0 \\ 0 & 0 & 2 & -2 \end{bmatrix}$$

and

$$F_4^{-1} = 1/4 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & (-i) & (-i)^2 & (-i)^3 \\ 1 & (-i)^2 & (-i)^4 & (-i)^6 \\ 1 & (-i)^3 & (-i)^6 & (-i)^9 \end{bmatrix}.$$

The essential point is that the inverse matrices have the same form as the originals. If we transform quickly, we can invert quickly - between coefficients and function values. The Fourier coefficients come from values at n points. The Haar coefficients come from values on n subintervals.

1.5 Application - data compression

An immediate application of the wavelets transform is in image compression. The overall procedure is to take the wavelet transform of a digitized image, and then to "allocate bits" among the wavelet coefficients in some

highly nonuniform, optimized, manner. In general, large wavelet coefficients get quantized accurately, while small coefficients are quantized coarsely with only a bit or two - or else are truncated completely. If the resulting quantization levels are still statically nonuniform, they may then be further compressed.

The compression technique is quite straightforward. We perform wavelet encoding with a simple truncation: We keep, with full accuracy, all wavelet coefficients larger than some threshold, and we delete (set to zero) all smaller wavelet coefficients. We can then adjust the threshold to vary the fraction of preserved coefficients.

Chapter 2

Implementing the Wavelet Transform in 1-D

2.1 Connection to the theory of filters

We describe how an arbitrary signal in the discrete infinite-dimensional vector space $L^2(\mathbf{Z})$ or the discrete N-dimensional vector space $\mathbf{C}^N$ (where we assume $2|N$) can be written as a weighed sum of certain "elementary" synthesizing functions. The expression of a signal x as a weighed sum of certain synthesizing functions is called a *decomposition* of x. In order to set up the problem consider the arrangement of filters, downsamples, and upsamples in Figure 2.1.

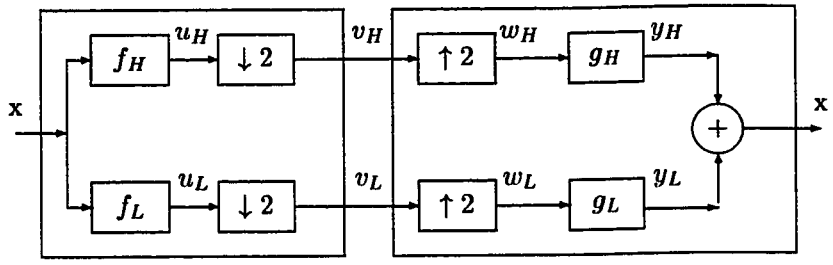


Figure 2.1

By "*downarrow*" is meant *downsampling,* or the deletion of every other number from the input sequence. By "*uparrow*" is meant *upsampling,* or the insertion of a zero between every pair of numbers in the input sequence. The symbols D and U will also be used to denote the down and upsampling operators. The symbols f_L and f_H denote *analyzing filters,* while g_L and g_H are *synthesizing filters.* The subscripts "H" and "L" could be thought of as abbreviations for "high-pass" and "low-pass," respectively, because it is usual, through not necessary, that f_H and g_H are high-pass filters, and f_L and g_L low-pass filters.

If the signal x lies in infinite-dimensional vector space $V = L^2(\mathbf{Z})$, then we will assume that f_L, f_H, g_L, g_H, $\in L^1(\mathbf{Z})$, the sequences u_L, v_L, w_L, y_L, u_H, v_H, w_H, and y_H all belong to $L^2(\mathbf{Z})$. For x in the finite-dimensional signal (vector) space $V = \mathbf{C}^N = L^2(\mathbf{Z}_N)$, we will assume $f_L, f_H, g_L, g_H \in \mathbf{C}^N$. Then for $x \in \mathbf{C}^N$ the sequences u_L, w_L, y_L, u_H, w_H, and y_H belong to $\mathbf{C}^N$, while the sequences v_L and v_H belong to $\mathbf{C}^{N/2}$.

The sequences v_H and v_L define a discrete orthonormal wavelet transform

of the signal x. The synthezing-filter sequences g_H and g_L will be chosen so as to attain a perfect reconstruction of x. Perfect reconstruction will be seen to define a decomposition of x.

2.2 Perfect reconstruction filters

For an infinite sequence $x = (..., x(-1), x(0), x(1), ...) \in L^2(\mathbf{Z}), x(i) \in \mathbf{C}$ for all i, the z-transform $\hat{x}(z)$ of x is written:

$$\hat{x}(z) = \sum_{n=-\infty}^{\infty} x(n) z^{-n}, \tag{2.1}$$

where the interminate z ranges over the unit circle $\mathbf{T}$ in the complex plane. In the standard definition of the z-transform, the interminate z ranges over the entire complex plane. Here we will use restrictions of the standard definition. Note that $x(n)$ is the nth Fourier coefficient of the function that maps $\omega \in [-\pi, \pi]$ to $\hat{x}(e^{-j\omega})$; in particular then,

$$x(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} \hat{x}(e^{-j\omega}) e^{-jn\omega} d\omega. \tag{2.2}$$

From (2.2), if $\hat{x}(z) = \hat{y}(z)$ for all $z \in \mathbf{T}$, then $x = y$; hence the z-transform is one-to-one. If x is a finite sequence $x = (x(0), ..., x(N-1)) \in \mathbf{C^N}$ for some $N \in \mathbf{Z^+}$, then the z-transform $\hat{x}(z)$ of x is written:

$$\hat{x}(z) = \sum_{n=0}^{N-1} x(n) z^{-n}. \tag{2.3}$$

Let R denote a *right-shift operator* that acts upon a sequence $x \in L^2(\mathbf{Z})$

or $x \in \mathbf{C}^N$ such that $(Rx)(n) = x(n-1)$, or, equivalently,

$$\widehat{(Rx)}(z) = z^{-1}\hat{x}(z). \tag{2.4}$$

In the finite-dimensional signal space $\mathbf{C}^N$, the right shift operator defined in (2.4) wraps sequences around. When x is shifted right once, then the number $x(N-1)$ moves to occupy the place where $x(0)$ was. This is a consequence of using $\mathbf{Z}_N$ arithmetic upon indices for x. The z-transform notation reflects this wrap-around accurately when the canonical form is employed.

Let f be the impulse response of a filter with input x and output y. Define $\tilde{f}(n) = \bar{f}(-n)$. The sequence $\tilde{f}$ is the complex conjugate of the time reversal of the sequence f. Then,

$$\begin{aligned} y(k) &= \sum_n x(n)f(k-n) = \sum_n x(n)\bar{\tilde{f}}(n-k) \\ &= \sum_n x(n)(R^k(\bar{\tilde{f}})(n) \\ &= \sum_n x(n)\overline{R^k\tilde{f}}(n) = \langle x, R^k\tilde{f}\rangle. \end{aligned} \tag{2.5}$$

From the low-pass (lower)branch of Figure 2.1 we have:

$$\hat{u}_L(z) = \hat{f}_L(z)\hat{x}(z) \tag{2.6}$$

$$\hat{v}_L(z) = \widehat{(\mathcal{D}u_L)}(z) = \frac{1}{2}(\hat{u}_L(z^{1/2}) + \hat{u}_L(-z^{1/2})) \tag{2.7}$$

$$\hat{w}_L(z) = \widehat{(\mathcal{U}v_L)}(z) = \hat{v}_L(z^2) \tag{2.8}$$

$$\hat{y}_L(z) = \hat{g}_L(z)\hat{w}_L(z). \tag{2.9}$$

Equations (2.7) and (2.8) describe the down and upsampling operations, respectively, and may be considered to define those operations. Alternatively, we may define the downsampling operator $\mathcal{D} : L^2(\mathbf{Z}) \to L^2(\mathbf{Z})$ or $\mathcal{D} : \mathbf{C}^N \to \mathbf{C}^{N/2}$ by

$$(\mathcal{D}u)(n) = u(2n), n \in \mathbf{Z} \text{ or } n \in \mathbf{Z}_{N/2}. \tag{2.10}$$

The definitions of $\mathcal{D}$ in (2.7) and (2.10) are equivalent because

$$\begin{aligned} \hat{u}(z^{1/2}) + \hat{u}(-z^{1/2}) &= \sum_n u(n) z^{-n/2} + \sum_n u(n)(-1)^n z^{-n/2} \quad (2.11) \\ &= 2 \sum_{n \text{even}} u(n) z^{-n/2} = 2 \sum_m u(2m) z^{-m}. \quad (2.12) \end{aligned}$$

Similarly, the upsampling operator $\mathcal{U} : L^2(\mathbf{Z}) \to L^2(\mathbf{Z})$ or $\mathcal{U} : \mathbf{C}^{N/2} \to \mathbf{C}^N$ may be defined as:

$$(\mathcal{U}v)(n) = \begin{cases} v(n/2), & n \text{ even} \\ 0, & \text{otherwise.} \end{cases} \tag{2.13}$$

The definitions of $\mathcal{U}$ in (2.8) and (2.13) are equivalent. From (2.6)-(2.9),

$$\hat{y}_L(z) = \frac{1}{2}\hat{g}_L(z)\hat{f}_L(z)\hat{x}(z) + \frac{1}{2}\hat{g}_L(z)\hat{f}_L(-z)\hat{x}(-z). \tag{2.14}$$

Similarly, from the high-pass branch in Figure 2.1,

$$\hat{y}_H(z) = \frac{1}{2}\hat{g}_H(z)\hat{f}_H(z)\hat{x}(z) + \frac{1}{2}\hat{g}_H(z)\hat{f}_H(-z)\hat{x}(-z). \tag{2.15}$$

Perfect reconstruction holds in Figure 2.1 if and only if $\hat{x}(z) = \hat{y}_L(z) + \hat{y}_H(z)$ other words if and only if

$$\begin{aligned} \hat{x}(z) &= \frac{1}{2}(\hat{g}_L(z)\hat{f}_L(z) + \hat{g}_H(z)\hat{f}_H(z))\hat{x}(z) + \\ &\quad \frac{1}{2}(\hat{g}_L(z)\hat{f}_L(-z) + \hat{g}_H(z)\hat{f}_H(-z))\hat{x}(-z). \end{aligned} \tag{2.16}$$

We also have the following: If $P(z), Q(z)$ are fixed polynomials in z and for all $\hat{x}(z)$,

$$\hat{x}(z) = P(z)\hat{x}(z) + Q(z)\hat{x}(-z), \tag{2.17}$$

then $P(z) = 1$ and $Q(z) = 0$. In order to prove this we substituting $\hat{x}(z) = 1$ and $\hat{x}(z) = z$ into (2.17) we have

$$z = zP(z) + zQ(z), \tag{2.18}$$

$$z = zP(z) - zQ(z). \tag{2.19}$$

By adding and subtracting (2.18) and (2.19) we have (2.17). From (2.16) and (2.17) we deduce that perfect reconstruction holds if and only if the following equations are true:

$$\hat{g}_L(z)\hat{f}_L(z) + \hat{g}_H(z)\hat{f}_H(z) = 2, \tag{2.20}$$

$$\hat{g}_L(z)\hat{f}_L(-z) + \hat{g}_H(z)\hat{f}_H(-z) = 0. \tag{2.21}$$

The system of equations (2.20) and (2.21) can be written as the single

matrix equation below.

$$\frac{1}{\sqrt{2}}\begin{pmatrix} \hat{f}_L(z) & \hat{f}_H(z) \\ \hat{f}_L(-z) & \hat{f}_H(-z) \end{pmatrix}\begin{pmatrix} \hat{g}_L(z) \\ \hat{g}_H(z) \end{pmatrix} = A(z)\begin{pmatrix} \hat{g}_L(z) \\ \hat{g}_H(z) \end{pmatrix} = \begin{pmatrix} \sqrt{2} \\ 0 \end{pmatrix}, \tag{2.22}$$

where $A(z)$ is

$$A(z) = \frac{1}{\sqrt{2}}\begin{pmatrix} \hat{f}_L(z) & \hat{f}_H(z) \\ \hat{f}_L(-z) & \hat{f}_H(-z) \end{pmatrix} \tag{2.23}$$

The condition (2.23) is necessary and sufficient for perfect reconstruction, *whether or not* $\tilde{B}$ is orthonormal. If $\tilde{B}$ orthonormal then the system matrix $A(z)$ is unitary and is, therefore, particularly easy to invert: $A^{-1}(z) = A^*(z) = \bar{A}^T(z)$. Hence (2.23) is easily solved:

$$\begin{pmatrix} \hat{g}_L(z) \\ \hat{g}_H(z) \end{pmatrix} = A^*(z)\begin{pmatrix} \sqrt{2} \\ 0 \end{pmatrix} = \begin{pmatrix} \bar{\hat{f}}_L(z) \\ \bar{\hat{f}}_H(z) \end{pmatrix} \tag{2.24}$$

Without proof we state that when $\tilde{B}$ is orthonormal, perfect reconstruction requires that $g_L = \tilde{f}_L$ and $g_H = \tilde{f}_H$. This is hardly a surprising result in view of that fact that given an orthonormal basis $\tilde{B}$ of an inner product space $V, x \in V$ can be decomposed as a weighted sum of vectors in $\tilde{B}$, where the weights are given by the inner products of x computed against the basis vectors. By perfect reconstruction in Figure 2.1,

$$x(n) \;=\; (g_L * (UD(x * f_L)))(n) + (g_H * (UD(x * f_H)))(n) \tag{2.25}$$

$$= \sum_k g_L(n-2k)\langle x, R^{2k}\tilde{f}_L\rangle + \sum_k g_H(n-2k)\langle x, R^{2k}\tilde{f}_H\rangle \quad (2.26)$$

$$= \sum_k \langle x, R^{2k}\tilde{f}_L\rangle (R^{2k}g_L)(n) + \sum_K \langle x, R^{2k}\tilde{f}_H\rangle (R^{2k}g_H)(n). \quad (2.27)$$

For $x \in \mathbf{C}^N$, the index k in (2.26) and (2.27) runs over $\mathbf{Z}_{N/2}$. In view of the decomposition (2.27) of x and the orthogonality of $\tilde{B}$, we can say that $g_L = \tilde{f}_L$ and $g_H = \tilde{f}_H$. Equation (2.27), along with the equations $g_L = \tilde{f}_L$ and $g_H = \tilde{f}_H$, demonstrates the *completeness* of $\tilde{B}$. These three equations show that every vector $x \in L^2(\mathbf{Z})$ or $\mathbf{C}^N$ can be written as a sum of the elements of $\tilde{B}$. The sequences $\tilde{f}_L$ and $\tilde{f}_H$ are *first-generation wavelets.*

We do not assume the orthonormality of $\tilde{B}$; it is possible for (2.25)-(2.27) to hold with $g_L \neq \tilde{h}_L$ and $g_H \neq \tilde{h}_H$. This is the first step in the developing of "bioorthogonal wavelets." [17], [19]

2.3 Daubechies Wavelets

We will construct orthonormal bases of wavelets, i.e., orthonormal bases $\{\psi_{jk}; j,k \in \mathbf{Z}\}$ for $L^2(\mathbf{R})$, where

$$\psi_{jk}(x) = 2^{j/2}\psi(2^j x - k) \quad (2.28)$$

for some (very particular!) $\psi \in L^2(\mathbf{R})$. The function (2.28) are *wavelets* because they are all generated from one single function by dilations and translations. The wavelets need not be orthogonal or even linearly independent [50], [51], [84]. Even the special case of orthonormal wavelets need not always be of the form (2.28). Basic dilation factors different from 2 are

possible: there exist orthonormal bases in which this factor is any rational $p/q > 1$; in more than one dimension we may even choose a dilation *matrix* instead of an isotropic dilation factor. In these more general cases, it may be necessary to introduce more than one ψ (but always a finite number). We shall restrict ourselves to one dimension here, and to the dilation factor 2, as in (2.28). Bases with factor 2 are by far the easiest to implement for numerical computations.

All interesting examples of orthonormal wavelet bases can be constructed via *multiresolution analysis* [78], [79], [54]. In this case the wavelet coefficients $\langle f, \psi_{jk} \rangle$ for fixed j describe the difference between two approximations of f, one with resolution 2^{j-1}, and one with the coarser resolution 2^j.

The successive approximation spaces V_j in a multiresolution analysis can be characterized by means of a scaling function ϕ. More precisely, we assume that the integer translates of ϕ are an orthonormal basis for the space V_0, which we define to be the approximation space with resolution 1. The approximation spaces V_j with resolution 2^j are then defined as the closed linear spans of the $\phi_{jk} (k \in \mathbf{Z})$, where

$$\phi_{jk} = 2^{-j/2} \phi(2^{-j}x - k). \tag{2.29}$$

To ensure that projections on the V_j describe successive approximation, we require $V_0 \subseteq V_1$, which implies

$$...V_{-2} \subseteq V_{-1} \subseteq V_0 \subseteq V_1 \subseteq V_2 \subseteq ... \tag{2.30}$$

This imposes a restriction on ϕ : since $\in V_0 \subseteq V_1 = span\{\phi_{1k}, k \in \mathbf{Z}\}$, there must exist c_n such that

$$\phi(x) = \sum_n c_n \phi(2x - n). \tag{2.31}$$

In order to have a complete description of $L^2(\mathbf{R})$, we also impose

$$\cap_{j \in \mathbf{Z}} V_j = \{0\}. \tag{2.32}$$

$$\cup_{j \in \mathbf{Z}} V_j = L^2(\mathbf{R}). \tag{2.33}$$

For every multiresolution analysis as described above, there exists a corresponding orthonormal basis of wavelets defined by

$$\psi(x) = \sum_{n \in \mathbf{Z}} (-1)^n c_{n+1} \phi(2x + n), \tag{2.34}$$

where c_n are the coefficients in (2.31). Then ψ_{0n} are then an orthonormal basis for the orthogonal complement W_0 of V_0 in V_1. This phenomenon repeats itself at every resolution level j. It follows that for every j, the $\langle f, \psi_{jk} \rangle$ determine the difference in information between the approximations $P_j f, P_{j-1} f$ at resolutions $2^j, 2^{j-1}$, respectively:

$$P_{j-1} f = P_j f + \sum_k \langle f, \psi_{jk} \rangle \psi_{jk} \tag{2.35}$$

Consequently, by (2.30), (2.32) and (2.33) the $(\psi_{jk}; j, k \in \mathbf{Z})$ constitute

an orthonormal basis for $L^2(\mathbf{R})$.

One advantage of the "nested" structure of a multiresolution analysis is that it leads to the tree-structured *pyramid algorithm* for the decomposition and reconstruction of functions. Instead of computing all the inner products $\langle f, \psi_{jk} \rangle$ directly, we proceed in a hierarchic way:

1 compute $\langle f, \phi_{jk} \rangle$ for the finest resolution level j wanted (if the data are given in a discrete fashion, then these discrete data can just be taken to be $\langle f, \phi_{jk} \rangle$);

2 then compute $\langle , \psi_{j+k} \rangle$ at the next finest resolution level by applying (2.31) and (2.39),

$$\langle f, \psi_{j+k} \rangle = \frac{1}{\sqrt{2}} \sum_n (-1)^n c_{n+2k+1} \langle f, \phi_{jn} \rangle, \tag{2.36}$$

$$\langle f, \phi_{j+k} \rangle = \frac{1}{\sqrt{2}} \sum_n c_{n-2k} \langle f, \phi_{jn} \rangle; \tag{2.37}$$

3 iterate until the coarsest desired resolution level is attained. The total complexity of this calculation is lower, despite the computation of the seemingly unnecessary $\langle f, \phi_{jk} \rangle$, than if the $\langle f, \psi_{jk} \rangle$ were computed directly.

This trail shows how to construct an orthonormal basis of wavelets from any "decent" function ϕ satisfying an equation of type (2.31). In general, constructions starting from a choice of ϕ lead to ϕ, ψ, which are not compactly supported. The construction can, however, also be viewed differently. The

Fourier transform of ϕ is

$$\hat{\phi}(\xi) = \left[\frac{1}{2}\sum_n e^{in\xi/2}\right]\hat{\phi}(\frac{\xi}{2}). \tag{2.38}$$

which implies

$$\hat{\phi}(\xi) = \left[\Pi_{j=1}^{\infty} m_0(2^{-j}\xi)\right]\hat{\phi}(0), \tag{2.39}$$

where $\hat{}$ denotes the Fourier transform,

$$\hat{\phi}(\xi) = \frac{1}{\sqrt{2\pi}}\int dx e^{ix\xi}\phi(x). \tag{2.40}$$

with $m_0(\xi) = \frac{1}{2}\sum_n c_n e^{in\xi}$, so that, up to normalization, ϕ is completely determined by the c_n. Fixing the c_n, therefore, also defines a multiresolution analysis. The c_n have to satisfy certain conditions. Combining $\langle \phi_{ok}, \phi_{oi}\rangle = \delta_{k,i}$ with (2.31) immediately leads to

$$\sum_n c_n c_{n+2k} = 2\delta_{k,0}, \tag{2.41}$$

where we have assumed, as we shall do in the sequel, that the c_n are real. In terms of $m_0(\xi)$, (2.41) can be rewritten as

$$|m_0(\xi)|^2 + |m_0(\xi+\pi)|^2 = 1. \tag{2.42}$$

To ensure that ϕ is well defined, the infinite product in (2.39) must converge,

which implies $m_0(0) = 1$ or

$$\sum_n c_n = 2. \tag{2.43}$$

It follows that ϕ is uniquely determined by (2.31), up to normalization, which we fix by requiring $\int dx\phi(x) = 1$. One can show that (2.42) implies that ϕ is in $L^2(\mathbf{R})$, but unfortunately (2.41) is not sufficient to guarantee orthonormality of the ϕ_{on}. A counterexample is $c_0 = c_3 = 1$, all other $c_n = 0$, which leads to $\phi(x) = \frac{1}{3}$ for $0 \leq x < 3, \phi(x) = 0$ otherwise. Such counterexamples are rare, however. If $N_2 - N_1 = 3$, then the example above, $c_0 = c_3 = 1$, is the only one.

If we exclude these thin sets of "bad" choices for c_n (which can be done by various means [78]), then we can build orthonormal bases of wavelets starting from the c_n. Once orthonormality of the ϕ_{0k} is established, all the rest follows easily. Formula (2.34) for ψ leads immediately to orthogonality of the ψ_{0l} and ϕ_{0k},

$$\langle \psi_{0l}, \phi_{0k} \rangle = \frac{1}{2} \sum_{n,m} (-1)^n c_{n+2l+1} c_{m+2k} \langle \phi_{ln}, \phi_{lm} \rangle = \frac{1}{2} \sum_n (-1)^n c_{n+2l+1} c_{n+2k} = 0. \tag{2.44}$$

The last equality follows from the substitution $n = -m + 2(k + l) + 1$ for the summation index n. Similar manipulations prove

$$\langle \psi_{0l}, \psi_{0k} \rangle = \delta_{k,l} \tag{2.45}$$

and

$$\sum_k [\langle f, \phi_{0k} \rangle \phi_{0k} + \langle f, \psi_{0k} \rangle \psi_{0k}] = \sum_n \langle f, \phi_{ln} \rangle \phi_{ln}. \tag{2.46}$$

It follows that both $\{\phi_{ln}; n \in \mathbf{Z}\}$ and $\{\phi_{0k}, \psi_{0k}; k \in \mathbf{Z}\}$ are orthonormal bases for V_l. (In other words, (2.41) ensures that (2.31) and (2.34) describe an orthonormal basis transformation.) It follows that $W_0 = span(\psi_{ok})$ is the orthogonal complement of V_0 in V_1, and hence that the $\{\psi_{jk}; j, k \in \mathbf{Z}\}$ constitute an orthogonal basis for $L^2(\mathbf{R})$.

Constructing ψ from the c_n rather than from ϕ has the advantage of allowing better control over the supports of ϕ and ψ. If $c_n = 0$ for $n < N_1, n > N_2$, then support $(\phi) \subseteq [N_1, N_2]$. This method was used to construct orthonormal bases of wavelets with compact support, and arbitrarily high preassigned regularity (the size of support increases linearly with the number continuous derivatives). These orthonormal basis functions and the associated multiresolution analysis have been tried out for several applications, ranging from image processing to numerical analysis. For some of these applications, variations on the scheme [27] were requested, emphasizing other properties.

Depending on the application several possible variations can be requested. The following are the most recurrent wish items.

(1) More symmetry: the functions ϕ, ψ in [27] are very asymmetric. Complete symmetry is incompatible with the orthonormal basis condition, but is less asymmetry possible ?

(2) Better frequency resolution: orthonormal bases with basic multiplication factor 2 correspond to frequency intervals of 1 octave. Is better possible (e.g., $\frac{1}{2}$ octave), without giving up compact support?

(3) More regularity: is better regularity than in [27] achievable for the

same support width?

(4) More vanishing moments: for a fixed support width $2N-1$, the ψ_N of [27] have the maximum number of vanishing moments. The functions ϕ_N do not satisfy any moment condition, except $\int dx\phi_N(x) = 1$. For numerical analysis applications, it may be useful to give up some zero moments of ψ in order to obtain zero moments for ϕ, i.e., to have

$$\begin{aligned} \int dx\phi(x) &= 1, \\ \int dx x^l \phi(x) &= 0, l = 1, ..., L, \\ \int dx x^l \psi(x) &= 0, l = 1, ..., L, \end{aligned} \tag{2.47}$$

The ϕ, ψ can be constructed in a way to have the advantage that inner products with smooth functions are particularly appealing:

$$\begin{aligned} \int dx\phi_{jk}(x)f(x) &= 2^{j/2}\int dx\phi(2^j(x-2^jk))f(x) \\ &= 2^{-j/2}f(2^{-j}k) + \text{correction terms in } f^{(L+1)} \end{aligned} \tag{2.48}$$

Use the Taylor expansion of around $2^{-j}k$; the second through $(L+1)th$ terms vanish because of (2.47). Moreover, if the $(L+1)th$ derivative of f is uniformly bounded, then the correction terms in this formula are of order $2^{-(L+1/2)j}$.

If the analyzing wavelet has compact support in $[1 - N/2, N/2]$ and the set of wavelets $\psi_{jk}(x)$ provide an orthonormal basis for $L^2(\mathbf{R})$. For N=2 the conditions (2.43) and (2.41) give

$$\begin{aligned} c_0 + c_1 &= 2, \\ c_0^2 + c_1^2 &= 2, \end{aligned} \tag{2.49}$$

with the solution $c_0 = c_1 = 1$. The scaling function is just the box function defined before and corresponding wavelet is the Haar wavelet.

The case $N = 4$ is more interesting. We now need to solve the equations

$$\begin{aligned} c_0 + c_1 + c_2 + c_3 &= 2, \\ c_0^2 + c_1^2 + c_2^2 + c_3^2 &= 2, \\ c_0 c_2 + c_1 c_3 &= 0. \end{aligned} \tag{2.50}$$

Setting

$$\begin{aligned} x &= c_0 + c_1, \\ y &= c_0 - c_1, \end{aligned} \tag{2.51}$$

$$\begin{aligned} w &= c_2 + c_3, \\ z &= c_2 - c_3, \end{aligned} \tag{2.52}$$

this equations reduce to

$$\begin{aligned} (y+z)^2 &= 0, \\ x+w &= 2. \end{aligned} \tag{2.53}$$

and

$$(w-1)^2 + y^2 = 1. \tag{2.54}$$

$$\begin{aligned} c_0 &= (1-\cos\theta+\sin\theta)/2, \\ c_1 &= (1-\cos\theta-\sin\theta)/2, \\ c_2 &= (1+\cos\theta-\sin\theta)/2, \\ c_3 &= (1+\cos\theta+\sin\theta)/2. \end{aligned} \tag{2.55}$$

As a check we note that for $\theta = \pi/2$ we recover Haar wavelet.

$$\begin{aligned} c_0 &= 1, \\ c_1 &= 0, \\ c_2 &= 0, \\ c_3 &= 1. \end{aligned} \tag{2.56}$$

What do these look like for other values of θ? We might try to impose some natural condition on the wavelet in order to pick out a unique solution. Since the wavelet already satisfies $\int \psi(x)dx = 0$, which leads to the additional

equation

$$-c - 2 + 2c_1 - 3c_0 = 0. \tag{2.57}$$

The unique solution with this further condition is then

$$\begin{aligned} c_0 &= \frac{1+\sqrt{3}}{4}, \\ c_1 &= \frac{3+\sqrt{3}}{4}, \\ c_2 &= \frac{3-\sqrt{3}}{4}, \\ c_3 &= \frac{1-\sqrt{3}}{4}. \end{aligned} \tag{2.58}$$

The resulting analyzing wavelet is denoted $\psi_4(x)$ and was first constructed by Daubechies as the second wavelet in an infinite family starting with the Haar wavelet (ψ_2) and having an increasing number of vanishing moments. The scaling equation itself gives us a recursive procedure for calculating the scaling function and hence the wavelet. Simply start with the box function as a first guess, $\phi_0(x)$, and for the next guess use the scaling equation:

$$\phi_1(x) = \sum_k \phi_0(2x - k). \tag{2.59}$$

Continuing this procedure quickly gives the scaling function to any desired accuracy. Two useful facts follow immediately from this procedure. The first is that the scaling function ϕ has support in $[0, N-1]$ if the non-zero coefficients are $c_0, \ldots c_{N-1}$. The second is that ϕ is orthogonal to its integer translates. To see this, note that it is obviously true for the box function,

and this property is preserved by the recursion using (2.41).

Consider the following transformation matrix acting on a column vector of data to its right:

$$\begin{bmatrix}
c_0 & c_1 & c_2 & c_3 & & & & & & & & \\
c_3 & -c_2 & c_1 & -c_0 & & & & & & & & \\
 & & c_0 & c_1 & c_2 & c_3 & & & & & & \\
 & & c_3 & -c_2 & c_1 & -c_0 & & & & & & \\
\vdots & \vdots & & & & & \ddots & & & & & \\
 & & & & & & & c_0 & c_1 & c_2 & c_3 & \\
 & & & & & & & c_3 & -c_2 & c_1 & -c_0 & \\
c_2 & c_3 & & & & & & & & c_0 & c_1 & \\
c_1 & -c_0 & & & & & & & & c_3 & -c_2 &
\end{bmatrix} \tag{2.60}$$

Here blank entries signify zeroes. Note the structure of this matrix. The first row generates one component of data convolved with the filter coefficients $c_0, ..., c_3$. So do the third, fifth, and other odd rows. If the even rows followed this pattern, offset by one, then the matrix would be a circulant, that is, an ordinary convolution that could be done by FFT methods. Note how the last two rows wrap around like convolutions with periodic boundary conditions.) Instead of convolving with $c_0, ..., c_3$, however, the even rows perform a *different* convolution, with coefficients $c_3, -c_2, c_1, -c_0$. The action of the matrix, overall, is thus to perform two related convolutions, then to decimate each of them by half (throw away half the values), and interleave

the remaining halves.

It is useful to think of the filter $c_0, ..., c_3$ as being a smoothing filter, call it H, something like a moving average of four points, even through the c's are not all positive. Then, because of the minus signs, the filter $c_3, -c_2, c_1, -c_0$, call it G, is *not* a smoothing filter. (In signal processing contexts, H and G are called *quadrature mirror filters.)* In fact, the cs are chosen so as to make G yield a *zero* response to a sufficiently smooth data vector. This is done by requiring the sequence $c_3, -c_2, c_1, -c_0$ to have a certain number of vanishing moments. When this is the case for p moments a set of wavelets is said to satisfy an "approximation condition of order p." This results in the output of H, decimated by half, accurately representing the data's "smooth" information. The output of G, also decimated, is referred to as the data's "detail" information.

For such a characterization to be useful, it must be possible to reconstruct the original data vector of length N from $N/2$ smooth or s-components and its $N/2$ detail or d-components. That is effected by requiring the matrix

(2.60) to be orthogonal, so that its inverse is just the transposed matrix

$$\begin{bmatrix}
c_0 & c_3 & & & \cdots & & & & & c_2 & c_1 \\
c_1 & -c_2 & & & \cdots & & & & & c_3 & -c_0 \\
c_2 & c_1 & c_0 & c_3 & & & & & & & \\
c_3 & -c_0 & c_1 & -c_2 & & & & & & & \\
 & & & & \ddots & & & & & & \\
 & & & & & c_2 & c_1 & c_0 & c_3 & & \\
 & & & & & c_3 & -c_0 & c_1 & -c_2 & & \\
 & & & & & & & c_2 & c_1 & c_0 & c_3 \\
 & & & & & & & c_3 & -c_0 & c_1 & -c_2
\end{bmatrix} \tag{2.61}$$

The matrix (2.61) is inverse to matrix (2.60) if and only if these two equations hold,

$$\begin{aligned} c_0^2 + c_1^2 + c_2^2 + c_3^2 &= 1, \\ c_2 c_0 + c_3 c_1 &= 0. \end{aligned} \tag{2.62}$$

(There is a change of scale here from usage in other parts of this thesis.) If additionally we require the approximation condition of order $p = 2$, then two additional relations are required,

$$\begin{aligned} c_3 - c_2 + c_1 - c_0 &= 0, \\ 0c_3 - 1c_2 + 2c_1 - 3c_0 &= 0. \end{aligned} \tag{2.63}$$

Equations (2.62), (2.63) are four equations for the four unknown $c_0, ..., c_3$. The unique solution is

$$\begin{aligned} c_0 &= \frac{(1+\sqrt{3})}{4\sqrt{2}}, \\ c_1 &= \frac{(3+\sqrt{3})}{4\sqrt{2}}, \\ c_2 &= \frac{(3-\sqrt{3})}{4\sqrt{2}}, \\ c_3 &= \frac{(1-\sqrt{3})}{4\sqrt{2}}. \end{aligned} \tag{2.64}$$

In fact DAUB4 is only the most compact of a sequence of wavelet sets: if we had six coefficients instead of four, there would be three orthogonality requirements in (2.62) (with offsets of zero, two, and four), and we could require the vanishing of $p = 3$ moments in (2.63). In this case, DAUB6, the solution coefficients can also be expressed in closed form,

$$\begin{aligned} c_0 &= (1+\sqrt{10}+\sqrt{5+2\sqrt{10}})/16\sqrt{2}, \\ c_1 &= (5+\sqrt{10}+3\sqrt{5+2\sqrt{10}})/16\sqrt{2}, \\ c_2 &= (10-2\sqrt{10}+2\sqrt{5+2\sqrt{10}})/16\sqrt{2}, \\ c_3 &= (10-2\sqrt{10}-2\sqrt{5+2\sqrt{10}})/16\sqrt{2}, \\ c_4 &= (5+\sqrt{10}-3\sqrt{5+2\sqrt{10}})/16\sqrt{2}, \\ c_5 &= (1+\sqrt{10}-\sqrt{5+2\sqrt{10}})/16\sqrt{2}. \end{aligned} \tag{2.65}$$

For higher p, up to 10, Daubechies [27] has tabulated the coefficients numerically. The number of coefficients increases by two each time p is increased by one.

2.4 Discrete Wavelet Transform (DWT)

The Discrete Wavelet Transform (DWT) consists of applying a wavelet coefficient matrix such as (2.60) *hierarchially,* first to the full data vector of length N, then to the vector of length $N/2$, then to the vector of length $N/4$, and so on until only trivial number of components (usually four) remain. This procedure called a *pyramidal algorithm,* for obvious reason. The output of the DWT consists of these remaining components and all the "detail" components that were accumulated along the way. A diagram

makes the procedure clear:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \\ y_{12} \\ y_{13} \\ y_{14} \\ y_{15} \\ y_{16} \end{bmatrix} \overrightarrow{daub} \begin{bmatrix} s_1 \\ d_1 \\ s_2 \\ d_2 \\ s_3 \\ d_3 \\ s_4 \\ d_4 \\ s_5 \\ d_5 \\ s_6 \\ d_6 \\ s_7 \\ d_6 \\ s_8 \\ d_8 \end{bmatrix} \overrightarrow{perm} \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \\ \underline{s_8} \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \end{bmatrix} \overrightarrow{daub} \begin{bmatrix} S_1 \\ D_1 \\ S_2 \\ D_2 \\ S_3 \\ D_3 \\ S_4 \\ \underline{D_4} \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \end{bmatrix} \overrightarrow{perm} \begin{bmatrix} S_1 \\ S_2 \\ S_3 \\ \underline{S_4} \\ D_1 \\ D_2 \\ D_3 \\ \underline{D_4} \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \end{bmatrix} \overrightarrow{daub} \begin{bmatrix} S_1 \\ S_2 \\ D_1 \\ D_2 \\ D_1 \\ D_2 \\ D_3 \\ \underline{D_4} \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \end{bmatrix} \tag{2.66}$$

If the length of the data vector were a higher power of 2, there would be more stages of applying DAUB and permuting. The endpoint will always be a vector with two Ss and a hierarchy of Ds, Ds, ds. Notice that once ds are generated, they simply propagate through to all subsequent stages.

A value d_i of any level is termed a "wavelet coefficient" of the original data vector; the final values $S_1, ..., S_4$ should strictly be called "mother-

function coefficients," although the term "wavelet coefficients" is often used loosely for both *ds* and final *Ss*. Since the full procedure is a composition of orthogonal linear operations, the whole DWT is itself an orthogonal linear operator.

To invert the DWT, one simply reverses the procedure, starting with the smallest level of the hierarchy and working in (2.66) from right to left. In process we use inverse matrix (2.61).

In Section 2.8 you will find **MATLAB** function **daub** which produce Daubechies coefficients. For example, to find coefficients for Daubechies 4 you run function daub(4) in **MATLAB** environment.

Also in Section 2.8 you will see routine, **wt1**, that performs the pyramid algorithm on some data vector, using Daubechies 4 wavelets. In order to reproduce wavelet functions we run inverse DWT on the unit vector. The vector $e_1 = \{1, 0, ..., 0\}$, $e_2 = \{0, 1, 0, ..., 0\}$ and so on. Figure 2.2a (Section 2.8) shows the first wavelet function. Figure 2.2b shows the second wavelet function, and so on. Figure 2.2k shows the result of performing the inverse DWT on the input vector $e_3 e_9$.

The **wt1**, that performs the pyramid algorithm on some data vector, using Daubechies 20 wavelets or Daubechies 12 differs only by calling function daub20 or daub12 correspondently. Figures 2.3a-2.3n and 2.4a-2.4g shows the corresponding wavelet functions.

2.5 Lemarie Wavelets

The Fourier transform of a set of wavelet coefficients c_j is given by

$$H(\omega) = \sum_j c_j e^{ij\omega}. \tag{2.67}$$

Here H is a function periodic in 2π, and it has the same meaning as before: it is the wavelet filter, now written in the Fourier domain. A very useful fact is that the orthogonality conditions for the cs (equations (2.62)) collapse to two simple relations in the Fourier domain,

$$\frac{1}{2}|H(0)|^2 = 1, \tag{2.68}$$

$$\frac{1}{2}[|H(\omega)|^2 + |H(\omega+\pi)|^2] = 1, \tag{2.69}$$

Likewise the approximation conditions of order p (equations (2.63)) has a simple formulation, requiring that $H(\omega)$ have a pth order zero at $\omega = \pi$, or (equivalently)

$$H^{(m)}(\pi) = 0\, m = 0, 1, ..., p-1. \tag{2.70}$$

It is thus relatively straight forward to invent quadrature mirror filter sets in the Fourier domain. We simply invent a function $H(\omega)$ satisfying equations (2.68)-(2.70). To find the actual C_js applicable to a data (or $s-$component) vector of length N, and with periodic wrap-around as in matrices (2.66) and

(2.61), we invert equation (2.67) by the discrete Fourier transform

$$C_j = \frac{1}{N}\sum_{k=0}^{N-1} H(\frac{2\pi k}{N})e^{-2\pi ijk/N} = \sum_{v=j+kN} c_v \tag{2.71}$$

The quadrature mirror filter G (reversed c_j's with alternating signs), incidentally, has the Fourier representation

$$G(\omega) = e^{-i\omega}H^*(\omega+\pi) \tag{2.72}$$

where the asterisk denotes complex conjugation.

In general the above procedure will not produce wavelets with compact support. The Daubechies wavelets, or other wavelets with compact support, are specially chosen so that $H(\omega)$ is a trigonometric polynomial with only a small number of Fourier components, guaranteeing that there will be only a small number of nonzero c_j's.

On the other hand, there is often no particular reason to demand compact support. Giving it up in fact allows the ready construction of relatively smother wavelets (higher values of p). Even without compact support, the convolution implicit in the matrix equation (2.60) can be done efficiently by FFT methods.

Lemarie's wavelet has p=4, does not have compact support, and is defined by the choice of $H(\omega)$,

$$H(\omega) = [2(1-u)^4\frac{315-420u+126u^2-4u^3}{315-420v+126v^2-4v^3}]^{1/2} \tag{2.73}$$

where

$$u \equiv \sin^2 \frac{\omega}{2} \quad v \equiv \sin^2 \omega \tag{2.74}$$

An informal description is that the quadrature mirror filter $G(\omega)$ deriving from equation (2.73) has the property that it gives identically zero when applied to any function whose odd-numbered samples are equal to the cubic spline interpolation of its even-numbered samples. Since this class of functions includes many very smooth members, it follows that $H(\omega)$ does a very good job of truly selecting a function's smooth information content.

2.6 The data compression problem

The data compression problem in 1D is very straight forward. Let's assume that the input data is one row of a two dimensional array of pixels with size 128 X 128. The main program (Section 2.8 , file **DWT.C**) reads data from text file **data.txt**. The data is one dimensional array of integers with size equal $2^7 = 128$. The range of integers is from 0 to 255. The 0 means absolutely black and 255 means absolutely white. The data is in fact one row of a two dimensional monochrome image. We will talk about images in the next chapter. On Figure 2.5a you can see uncompressed input data. On Figures 2.5b-2.5g you can see two graphs: initial uncompressed vector and reproduction of initial vector from compressed data. The compression index ranges from 46.9% down to 3.9%. In main program we use **wt1** function to get DWT coefficients. The **wt1** function uses **daub4** function. After receiving DWT coefficients we sort them according their absolute values

using **insertsort** function. Then it is we have to remember position of each entry in uncompressed vector. Next we truncate sorted array keeping only the largest coefficients. Index 46.9% corresponds to 60 coefficients out of 128, and 3.9% to 5.

2.7 The data extension problem

Assume that the length of the signal $x(n)$ is of length of M. The input signal is extended by adding at least $\frac{1}{2}M$ values at each end. The total length must be power of 2. The following five nonpredictive ways of extending the finite signal have been investigated (Figure 2.6i-2.6v).

(i). Pad with zeros: where the signal is assumed to be zero outside its support, so that

$$\begin{aligned} x(n) &= 0 \text{ for } n < 0 \\ &\quad \text{and} n > M \end{aligned}$$

(ii). Circular extension: where the signal is assumed to be periodic with period M. Thus,

$$\begin{aligned} x(n) &= x((n+M) mod M) \text{ for } n < 0 \text{ and} \\ x(n) &= x(n mod M) \text{ for } n > M \end{aligned}$$

(iii). Replication of boundary values, which means that the signal is made continuous at the ends by repeating the first and the last values of the signal, respectively.

$$x(n) = x(0) \text{ for } n < 0 \text{ and}$$
$$x(n) = x(M) \text{ for } n > M$$

(iv). Symmetric extension [101], which is similar to (ii) by making the signal periodic, but this time by period 2M. This is achieved by extending the signal by its mirror image, whereby it becomes symmetric around the boundaries. Thus,

$$x(n) = x(-n) \text{ for } n < 0 \text{ and}$$
$$x(n) = x(2M - n) \text{ for } n > M$$

(v). Doubly symmetric extension, which is achieved by flipping the signal not only in time as (iv), but also in amplitude, as shown in Figure 2.6v. This is given by

$$x(n) = 2x(0) - x(-n) \text{ for } n < 0 \text{ and}$$
$$x(n) = 2x(M) - x(2M - n) \text{ for } n > M$$

Note that method (i) and (ii) will in general create a discontinuity in the extended signal, whereas (iii), (iv) and (v) will maintain continuity. For

method (v), the first derivative is continuous at the boundary as well. Note that the methods (iv) and (v) need a more complex implementation than the other tree. The complexity can, roughly speaking, be considered zero for methods (i)-(iii) (replication of zeros, circular addressing, and replication of first and last values, respectively). For method (iv) an address calculation is necessary to replicate the correct values. Finally, method (v) requires one addition and one multiplication, or shift operation, as well as the address calculation. We calculated the relative error (we used norm two) for each case from Figure 6.2. The results are: $\Delta_i = 0.006596$, $\Delta_{ii} = 0.008078$, $\Delta_{iii} = 0.009033$, $\Delta_{iv} = 0.007375$, $\Delta_v = 0.009033$.

Methods of signal extension

pad with zeroes

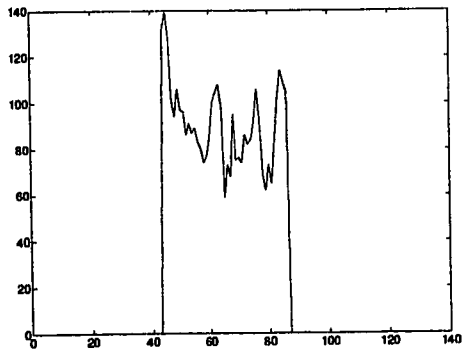


Figure 2.6i

circular extension

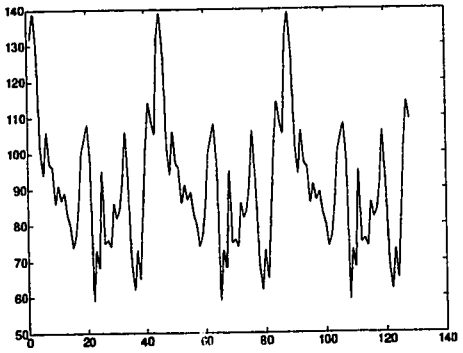


Figure 2.6ii

replication of edge values

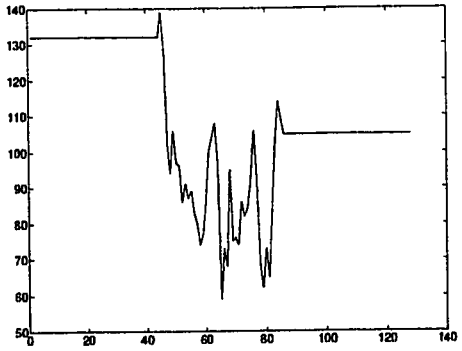


Figure 2.6iii

symmetric extension

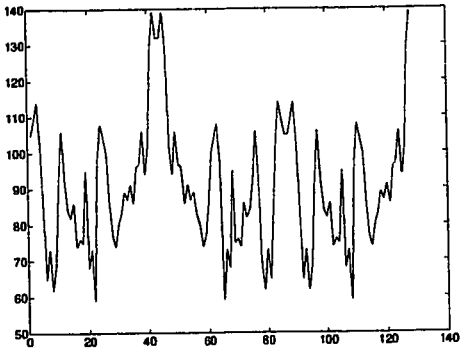


Figure 2.6 iv

doubly symmetric extension

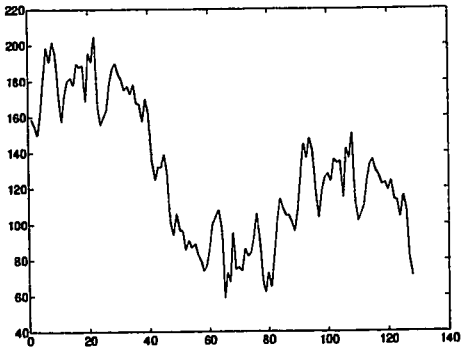


Figure 2.6 v

2.8 Computer Programs and Graphs in 1-D.

```
%       File daub.m
%       function h = daub(Nh)
%
%       Generate filter coefficients for the Daubechies orthogonal wavelets.
%
%       h = filter coefficients of Daubechies orthonormal compactly supported
%           wavelets
%       Nh = length of filter.

function h = daub(Nh)
K = Nh/2;
L = Nh/2;
N = 512;                                    % Use a 512 point FFT by default.
k = 0:N-1;

z = exp(j*2*pi*k/N);
tmp1 = (1 + z.^(-1)) / 2;
tmp2 = (1 - z.^(-1)) / 2;
Mz1 = zeros(1,N);
for l = 0:K-1
  Mz1 = Mz1 + binomial(L+l-1,l) * (-z).^l .* tmp2.^(2*l);
```

```
end
Mz1 = 4 * Mz1;

Mz1hat = log(Mz1);
m1hat = ifft(Mz1hat);

m1hat(N/2+1:N) = zeros(1,N/2);
m1hat(1) = m1hat(1) / 2;

G = exp(fft(m1hat,N));

Hz = G .* tmp1.^L;
h = real(ifft(Hz));
h = h(1:Nh)';

%File binomial.m
function h = binomial(N,M)
h = prod(1:N)/prod(1:M)/prod(1:(N-M));
```

```
//File Daub4.c
//Reproduction of Daub4 wavelets. Use unit vectors e1={1,0,...,0}
//e2={0,1,0,...,0}, etc. and apply inverse transform
#include <stdio.h>
#include <math.h>
#define MAX   1024   // maximum size of input vector
#define C0    0.4829629131445341 // Daubechies 4-coefficients
#define C1    0.8365163037378079
#define C2    0.2241438680420134
#define C3   -0.1294095225512604
typedef struct {
        double p;       // input vector
        int index;      // index of element in vector
   } entry;
void daub4(double[], int, int);
void wt1(double[], int, int);
void main()
{
   int n,i,j;       // n is power of two. The length of input vector
   double el;       // element of input vector
   double b[MAX];   // size of array of input vector equal MAX
   double  bb[MAX]; // intermediate copy of b[MAX]
   FILE *ifp, *ofp;
   entry ar[MAX];   // array of sorted DWT coefficients
```

```
  ifp = fopen("unit.txt", "r");  // open for reading
  ofp = fopen("unit_out.m", "w");//open for writing DWT coefficients
  fscanf(ifp, "%d", &n );
  for (i = 0; i < n; ++i){
      fscanf(ifp, "%lf", &el);
      b[i] = el;
  }
  fclose(ifp);
  wt1(b,n, -1);                    // obtaining DWT coefficients
  for(i = 0; i < n; ++i){
      fprintf(ofp, "%lf  \n", b[i]);
  }
  fclose(ofp);
}
```

```
void daub4(double a[], int n, int isign)
// Applies Daubechies 4-coefficient wavelet
// filter to data vector a[0..n-1]
// (for isign=1) or applies its transpose (for isign = -1).
{
  int nh1,i,j,nh;
  double wksp[MAX];
  if (n<4) return;
  nh=n/2;
  nh1=nh+1;
  if (isign>=0) {
    i=1;
    for(j=1;j<=n-3;j+=2){          /* Apply filter            */
      wksp[i-1]=C0*a[j-1]+C1*a[j]+C2*a[j+1] + C3*a[j+2];
      wksp[(i+nh)-1]=C3*a[j-1]-C2*a[j]+C1*a[j+1]-C0*a[j+2];
      i++;
    }
    wksp[i-1]=C0*a[n-2]+C1*a[n-1]+C2*a[0]+C3*a[1];
    wksp[(i+nh)-1]=C3*a[n-2]-C2*a[n-1]+C1*a[0]-C0*a[1];
  }else{                           /* Apply transpose filter   */
    wksp[0]=C2*a[nh-1]+C1*a[n-1]+C0*a[0]+C3*a[nh1-1];
    wksp[1]=C3*a[nh-1]-C0*a[n-1]+C1*a[0]-C2*a[nh1-1];
    j=3;
    for(i=1;i<=nh-1;i++){
```

```
      wksp[j-1]=C2*a[i-1]+C1*a[(i+nh)-1]+C0*a[i]+C3*a[(i+nh1)-1];
      wksp[j]=C3*a[i-1]-C0*a[(i+nh)-1]+C1*a[i]-C2*a[(i+nh1)-1];
      j+= 2;
    }
  }
  for(i=0;i<n;i++){
    a[i]=wksp[i];
  }
}

void wt1(double a[],int n,int isign)
/* One-dimensional discrete wavelet transform. This routine implements */
/* the pyramid algorithm, replacing a[0..n-1] by its wavelet transform */
/* (for isign=1), or performing the inverse operation (for isign=-1).  */
/* n must be power of 2.                                               */
{
  int nn;
  if (n<4) return;
  if (isign>=0) {          /* Wavelet transform. */
    nn=n;
    while(nn>=4) {
      daub4(a,nn,isign);
      nn/= 2;
    }
```

```
  }else{                    /* Inverse wavelet transform. */
    nn=4;
    while(nn<=n) {
      daub4(a,nn,isign);
      nn*= 2;
    }
  }
}
```

Daubechies 4 e_1

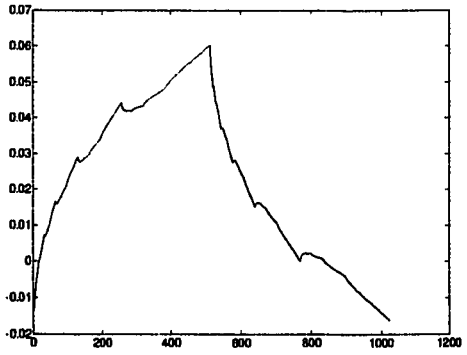


Figure 2.2a

Daubechies 4 e_2

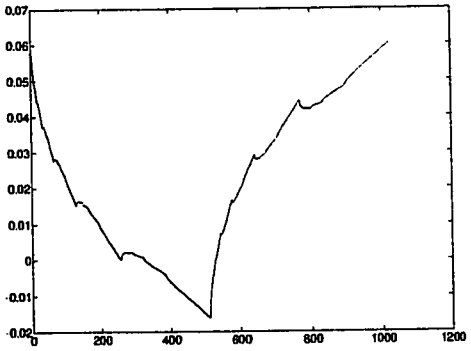


Figure 2.2b

Daubechies 4 e_3

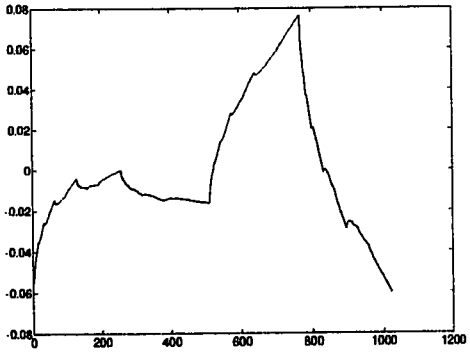


Figure 2.2c

Daubechies 4 e_4

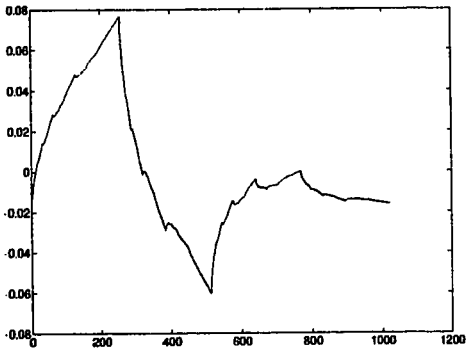


Figure 2.2d

Daubechies 4 e_5

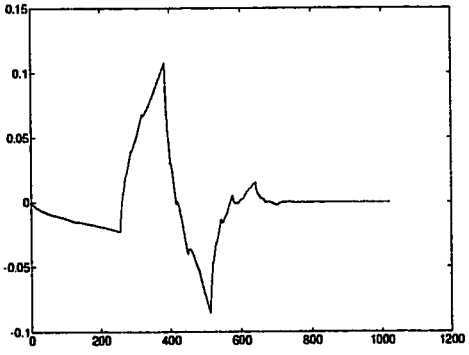


Figure 2.2e

Daubechies 4 e_6

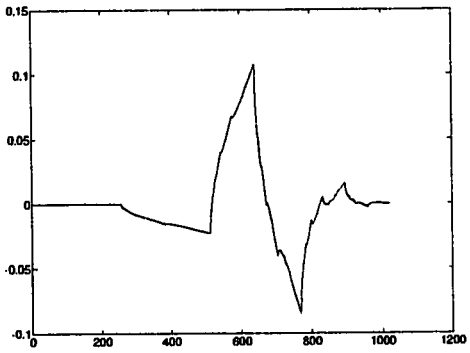


Figure 2.2f

Daubechies 4 e_9

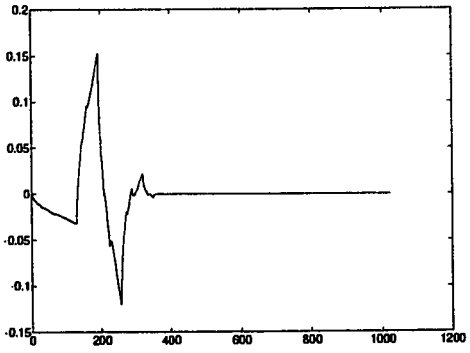


Figure 2.2g

Daubechies 4 e_{16}

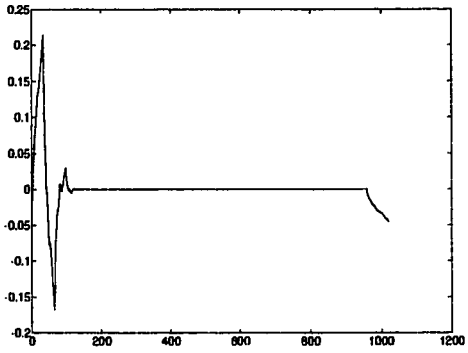


Figure 2.2h

Daubechies 4 e_{33}

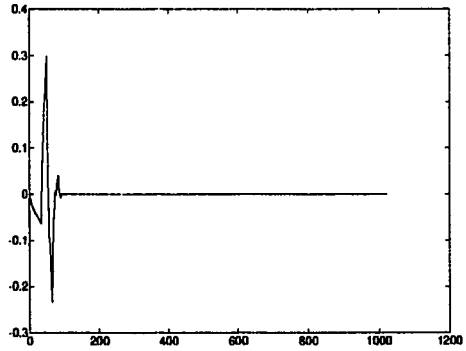


Figure 2.2i

Daubechies 4 e_{1024}

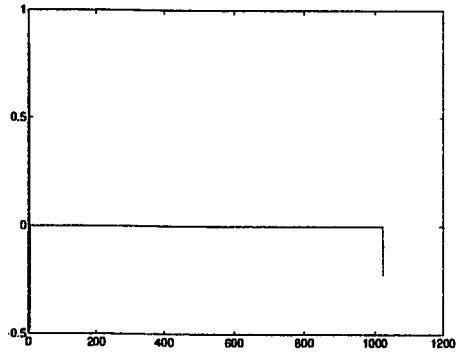


Figure 2.2j

Daubechies 4 $e_3 + e_9$

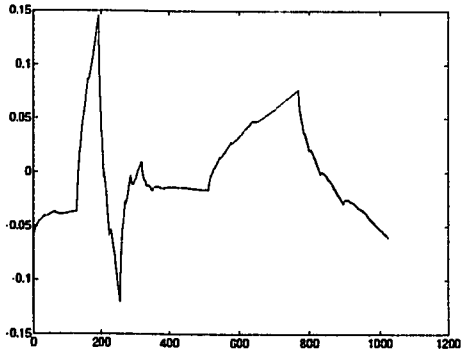


Figure 2.2k

Daubechies 4 e_{24}

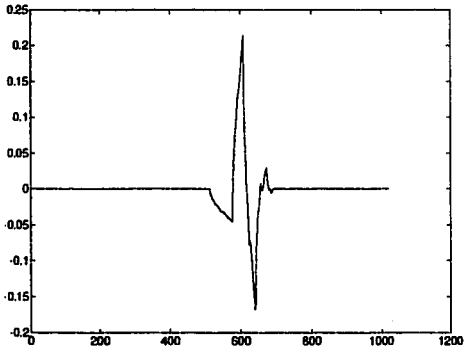


Figure 2.2l

Daubechies 4 e_{51}

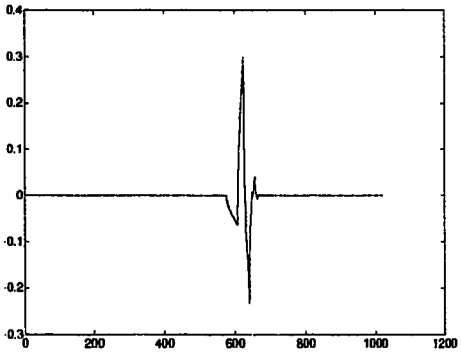


Figure 2.2m

Daubechies 4 e_{321}

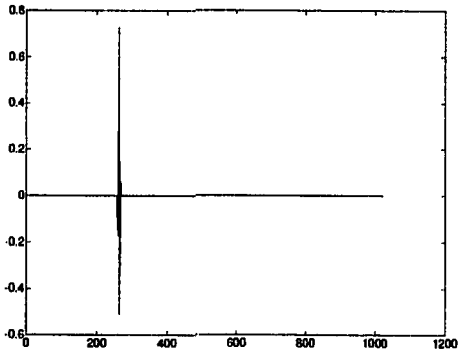


Figure 2.2n

```
//File Daub20.c
//Reproduction of Daub20 wavelets. Use unit vectors e1={1,0,...,0}
//e2={0,1,0,...,0}, etc. and apply inv. transform
#include <stdio.h>
#include <math.h>

#define MAX   1024   /* maximum size of input vector          */
double c20[21]={      /* Daubechies 20-coefficients */
0.0,
0.026670057901,
0.188176800078,
0.527201188932,
0.688459039454,
0.281172343661,
-0.249846424327,
-0.195946274377,
0.127369340336,
0.093057364604,
-0.071394147166,
-0.029457536822,
0.033212674059,
0.003606553567,
-0.010733175483,
0.001395351747,
```

```
0.001992405295,
-0.000685856695,
-0.000116466855,
0.000093588670,
-0.000013264203
};


typedef struct {
        double p;       /* input vector                              */
        int index;      /* index of element in vector                */
   } entry;

void daub20(double[], int, int);
void wt1(double[], int, int);

void main()
{
   int n,               /* power of two. The length of input vector  */
       i, j;
   double el;           /* element of input vector                   */
   double b[MAX];       /* size of array of input vector equal MAX   */
   double  bb[MAX];     /* intermediate copy of b[MAX]               */
   FILE *ifp, *ofp;
```

```
    entry ar[MAX];       /* array of sorted DWT coefficients          */

    ifp = fopen("unit.txt", "r");     /* open for reading */
    ofp = fopen("unit_out.m", "w");  /* open for writing DWT coefficients */
    fscanf(ifp, "%d", &n );
    for (i = 0; i < n; ++i){
        fscanf(ifp, "%lf", &el);
        b[i] = el;
    }
    fclose(ifp);
    wt1(b,n, -1);            /* obtaining DWT coefficients */

    for(i = 0; i < n; ++i){
        fprintf(ofp, "%lf  \n", b[i]);
    }
    fclose(ofp);
}

void daub20(double a[], int n, int isign)
{
    double ai,ai1,wksp[MAX];
    double sig = -1.0;
    double c20r[21];
    int i,ii,j,jf,jr,k,n1,ni,nj,nh,nmod;
```

```
int ioff,joff;


if(n < 4)
   return;
for(k=1; k <= 20;k++){
   c20r[21-k]=sig*c20[k];
   sig = -sig;
}
ioff =joff = -(n>>1);
nmod = 20*n;
n1=n-1;
nh = n>>1;
for(j=1;j<=n;j++)
   wksp[j]=0.0;
if(isign >= 0) {
  for(ii=1,i=1;i<=n;i+=2,ii++){
     ni=i+nmod+ioff;
     nj=i+nmod+joff;
     for(k=1;k<=20;k++){
        jf=n1 & (ni+k);
        jr=n1 & (nj+k);
        wksp[ii] +=c20[k]*a[jf+1-1];
        wksp[ii+nh] +=c20r[k]*a[jr+1-1];
     }
```

```
    }
  }
  else{
    for(ii=1,i=1;i<=n;i+=2,ii++){
      ai=a[ii-1];
      ai1=a[ii+nh-1];
      ni=i+nmod+ioff;
      nj=i+nmod+joff;
      for(k=1;k<=20;k++){
        jf=(n1 & (ni+k))+1;
        jr=(n1 & (nj+k))+1;
        wksp[jf] +=c20[k]*ai;
        wksp[jr] +=c20r[k]*ai1;
      }
          }
    }
    for(j=1;j<=n;j++)
          a[j-1]=wksp[j];
    free(wksp);
}//end daub20
```

Daubechies 20 e_1

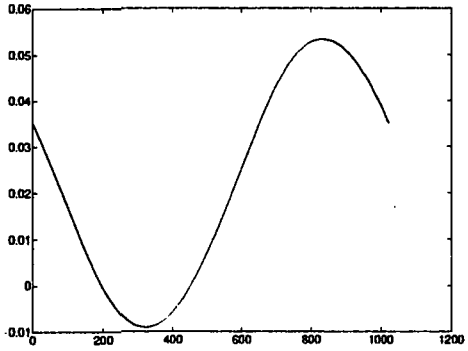


Figure 2.3a

Daubechies 20 e_2

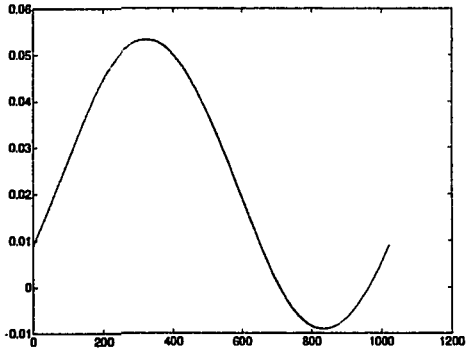


Figure 2.3b

Daubechies 20 e_3

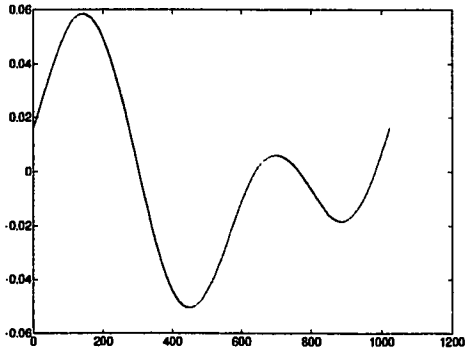


Figure 2.3c

Daubechies 20 e_4

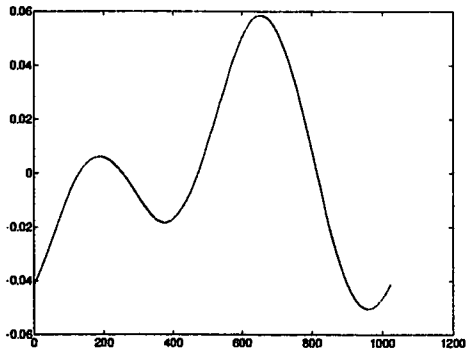


Figure 2.3d

Daubechies 20 e_5

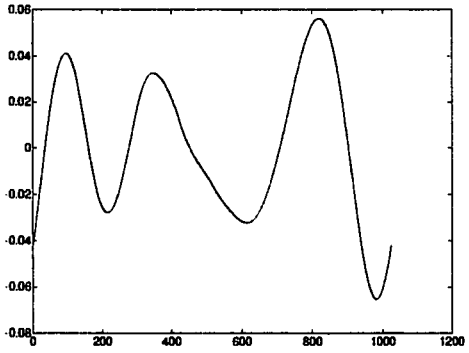


Figure 2.3e

Daubechies 20 e_6

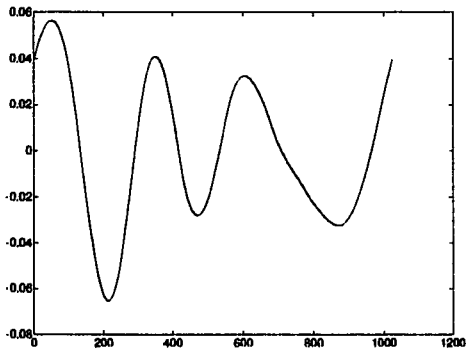


Figure 2.3f

Daubechies 20 e_{16}

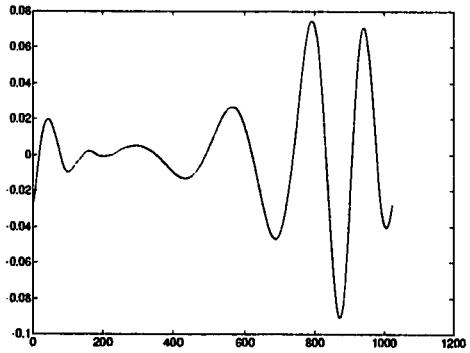


Figure 2.3g

Daubechies 20 e_{24}

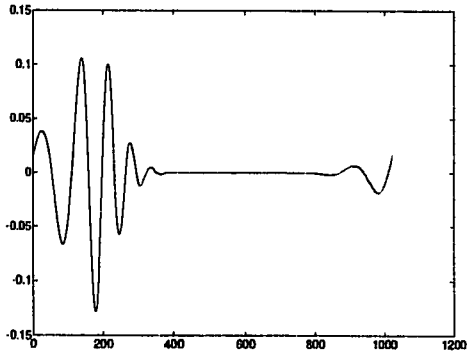


Figure 2.3h

Daubechies 20 e_{33}

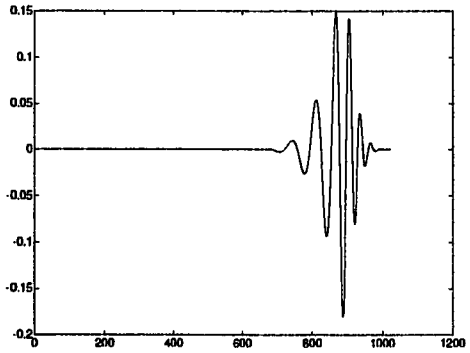


Figure 2.3i

Daubechies 20 e_{321}

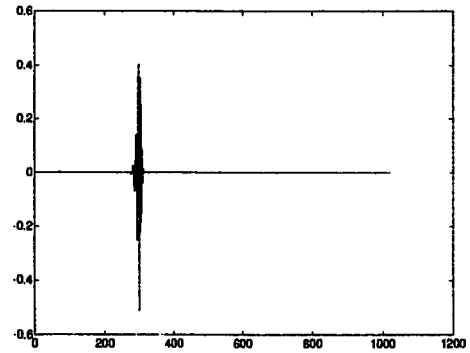


Figure 2.3j

Daubechies 20 e_{51}

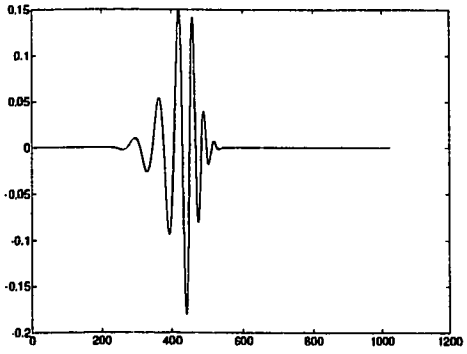


Figure 2.3k

Daubechies 20 e_{1024}

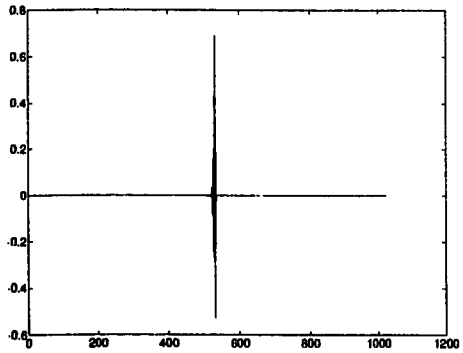


Figure 2.3l

Daubechies 20 $e_{64} + e_{129}$

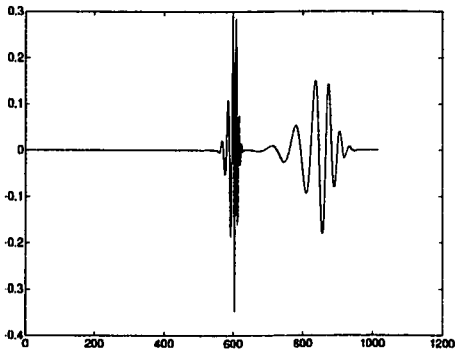


Figure 2.3m

Daubechies 20 $e_{259} + e_{480}$

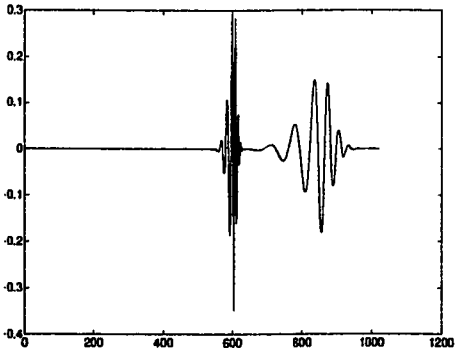


Figure 2.3n

```
//File Daub12.c
//Reproduction of Daub12 wavelets. Use unit vectors e1={1,0,...,0}
//e2={0,1,0,...,0}, etc. and apply inv. transform
//

#include <stdio.h>
#include <math.h>

#define MAX   1024   /* maximum size of input vector          */
double c12[13]={     /* Daubechies 12-coefficients */
0.0,
0.111540743350,
0.494623890398,
0.751133908021,
0.315250351709,
-0.226264693965,
-0.129766867567,
0.097501605587,
0.027522865530,
-0.031582039318,
0.000553842201,
0.004777257511,
-0.001077301085
};
```

```
typedef struct {
        double p;       /* input vector                              */
        int index;      /* index of element in vector                */
   } entry;

void daub12(double[], int, int);
void wt1(double[], int, int);

void main()
{
   int n,               /* power of two. The length of input vector  */
       i, j;
   double el;           /* element of input vector                   */
   double b[MAX];       /* size of array of input vector equal MAX   */
   double  bb[MAX];     /* intermediate copy of b[MAX]               */
   FILE *ifp, *ofp;
   entry ar[MAX];       /* array of sorted DWT coefficients          */

   ifp = fopen("unit.txt", "r");      /* open for reading */
   ofp = fopen("unit_out.m", "w");  /* open for writing DWT coefficients */
   fscanf(ifp, "%d", &n );
   for (i = 0; i < n; ++i){
```

```
        fscanf(ifp, "%lf", &el);
        b[i] = el;
    }
    fclose(ifp);
    wt1(b,n, -1);              /* obtaining DWT coefficients */
    for(i = 0; i < n; ++i){
        fprintf(ofp, "%lf  \n", b[i]);
    }
    fclose(ofp);
}


void daub12(double a[], int n, int isign)
{
    double ai,ai1,wksp[MAX+1];
    double sig = -1.0;
    double c12r[13];
    int i,ii,j,jf,jr,k,n1,ni,nj,nh,nmod;
    int ioff,joff;

    if(n < 4)
       return;
    for(k=1; k <= 12;k++){
       c12r[13-k]=sig*c12[k];
```

```
   sig = -sig;
}
ioff =joff = -(n>>1);
nmod = 12*n;
n1=n-1;
nh = n>>1;
for(j=1;j<=n;j++)
   wksp[j]=0.0;
if(isign >= 0) {
  for(ii=1,i=1;i<=n;i+=2,ii++){
     ni=i+nmod+ioff;
     nj=i+nmod+joff;
     for(k=1;k<=12;k++){
        jf=n1 & (ni+k);
        jr=n1 & (nj+k);
        wksp[ii] +=c12[k]*a[jf+1-1];
        wksp[ii+nh] +=c12r[k]*a[jr+1-1];
     }
  }
}
else{
   for(ii=1,i=1;i<=n;i+=2,ii++){
      ai=a[ii-1];
      ai1=a[ii+nh-1];
```

```
            ni=i+nmod+ioff;
            nj=i+nmod+joff;
            for(k=1;k<=12;k++){
               jf=(n1 & (ni+k))+1;
               jr=(n1 & (nj+k))+1;
               wksp[jf] +=c12[k]*ai;
               wksp[jr] +=c12r[k]*ai1;
            }
         }
      }
      for(j=1;j<=n;j++)
      a[j-1]=wksp[j];
}//end daub12
```

Daubechies 12 e_5

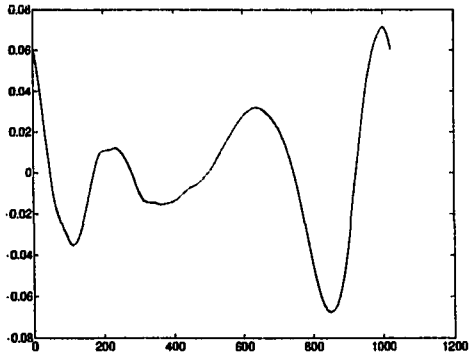


Figure 2.4a

Daubechies 12 e_{24}

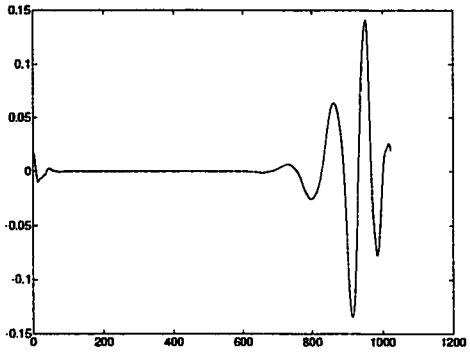


Figure 2.4b

Daubechies 12 e_{32}

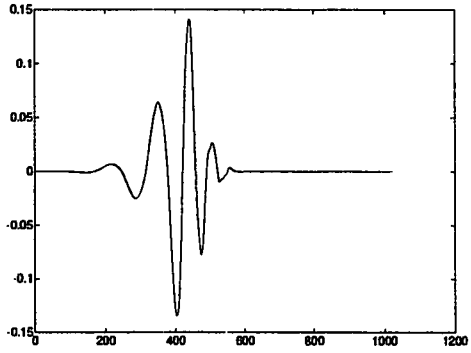


Figure 2.4c

Daubechies 12 e_{51}

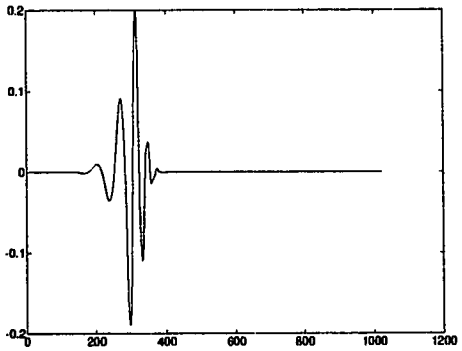


Figure 2.4d

Daubechies 12 e_{65}

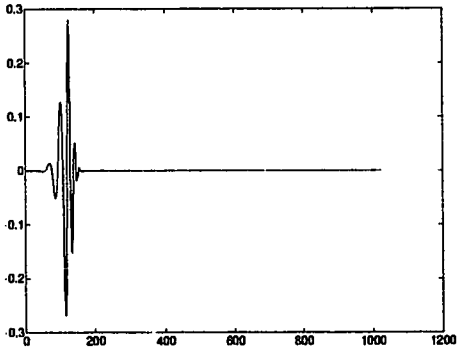


Figure 2.4e

Daubechies 12 e_{193}

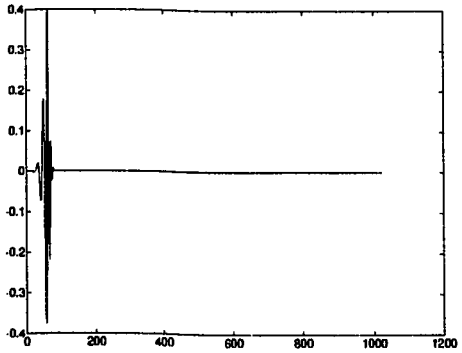


Figure 2.4f

Daubechies 12 $e_{642} + e_{60}$

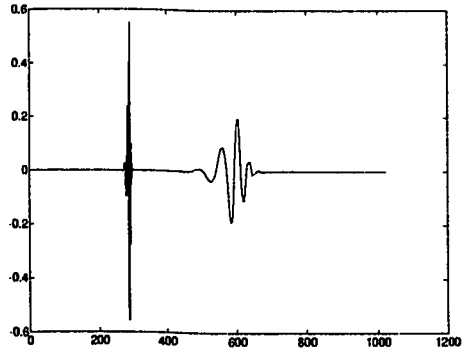


Figure 2.4g

1D uncompressed data

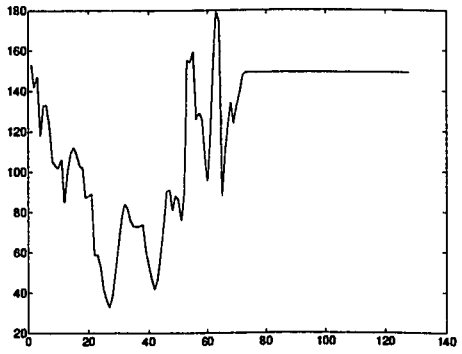


Figure 2.5a

1D compression 46.9%

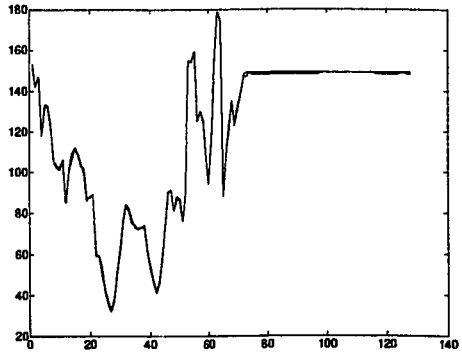


Figure 2.5b

1D compression 31.3%

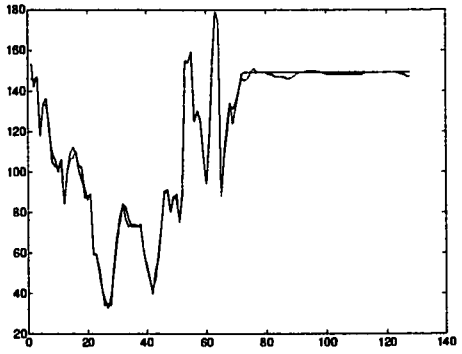


Figure 2.5c

1D compression 15.6%

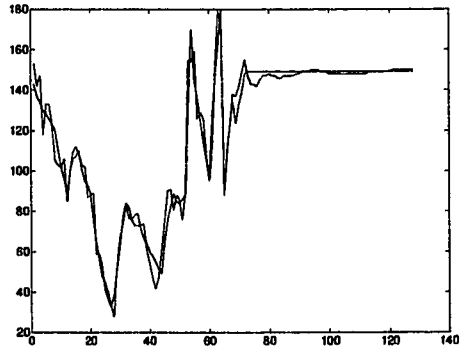


Figure 2.5d

1D compression 11.7%

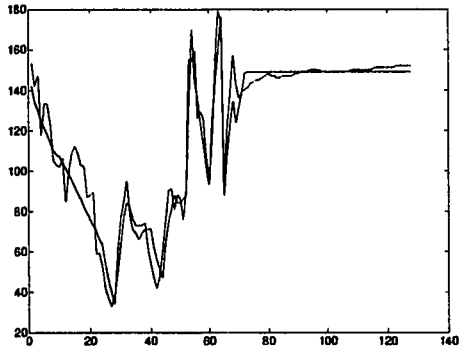


Figure 2.5e

1D compression 7.8%

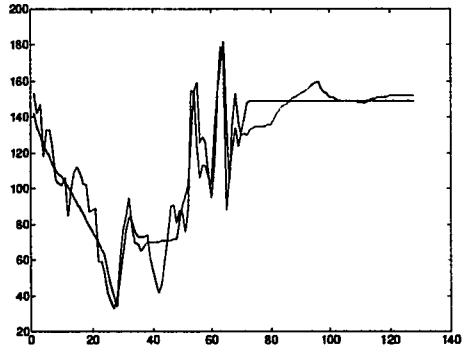


Figure 2.5f

1D compression 3.9%

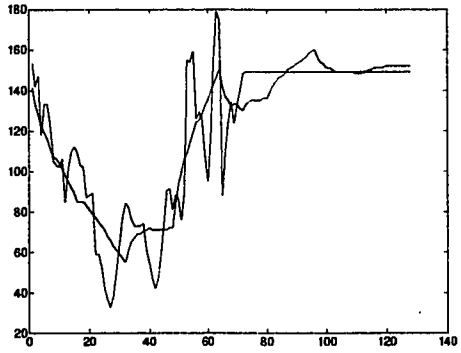


Figure 2.5g

Chapter 3

Multidimensional wavelet transform.

3.1 The transform in n dimensions.

A wavelet transform of a $n-$dimensional array is most easily obtained by transforming the array sequentially on its first index (for all values of its other indexes), then on its second, and so on. Each transformation corresponds to multiplication by an orthogonal matrix. By matrix associativity, the result is independent of the order in which the indices were transformed. The situation is exactly like that for multidimensional FFT's.

3.2 The two dimensional transform. Image compression.

A two-dimensional signal may be represented by a matrix of sampled values, where the sampling rates in each dimension are constant but not necessarily identical. A two-dimensional signal will be referred as an image, even for the case of a signal in one space and one time dimension. The wavelet treatment of such signals is the same in either case. Also, the samples will be referred to as picture elements, or "pixel." In our program we deal with monochrome black and white image with size 128 X 128. The digital representation of this is 0 for absolute black 255 for absolute white. The 2-d generalization of the wavelet analysis to study images is simple. One may simply perform wavelet transforms in one dimension, say in x, for each fixed value of y. Another possible choice is to transform in y at each fixed x. Or one may get fancy and parameterize the image space in terms of neighboring spirals or some other unbroken path, with all the pixels included in order to not omit any information. The goal of the analysis helps to determine which manner one should choose for transforming the image. The two-dimensional wavelet transform is always formed of independent one-dimensional transforms. The set of pixels which are used together for any individual wavelet transform referred as a "line of transformation." There will be a set of wavelet coefficients for each of these lines, so any code that performs a wavelet analysis on an image must keep track of an extra parameter, a "line parameter," that describes which set corresponds to which

line. To reconstruct an image, one we perform the inverse wavelet transform for each individual line of transformation. Therefore, it is necessary for the reconstruction algorithm to read the line parameters with the corresponding coefficients and to reproduce the image according to the same lines used for wavelet decomposition. In our example we compiled the program on 486DX machine with 16MB RAM. In order to do this we were forced to partition our 128 X 128 image to four 64 X 64 quadrants. We tried to compile on machine with 8MB RAM, but were out of memory. But even with 16MB RAM we had to pay very close attention to the memory management.

Wavelets are useful for reducing data sets, so they can be used for compressing 2-d images. Since each line of transformation usually can be reduced to a smaller data set, the overall image will be reduced. And the compression ratio is defined as the size of the initial data file divided by the size of the final data file after compression.

We tested our program on a two dimensional monochrome image. In Figures 3.1a -3.1n, section 3.3 you can see reconstructed images of "Clair" with different number of retained coefficients. We started with 100 % down to 1 %. The first recognizable differences occur when we retain only 50 % of the wavelet coefficients. Retaining only 1% of coefficients still gives us a recognizable image.

3.3 Computer Programs and Graphs in 2-D.

```
//TESTIM.C: Tests a variety of the image processing tools in IMTOOLS.C.
//Compile with the large memory model. Link with the files GRPHICS.C and
//IMTOOLS.C

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <alloc.h>
#include <string.h>
#include "grphics.h"
#include "pcxtool.h"
#include "imtools.h"

#define NMAX 128*128
void message(char *);

typedef struct {
        double p;       /* input vector                          */
        int index;      /* index of element in vector            */
}   entry;

#define SCREENHT 200   //The pixel height of the screen
```

```
#define SCREENWD 320   //The pixel width of the screen
#define NSTACK    50
#define M        100

FILE *Outfp;
FILE *Infp;
IMAGE Im1, Im;
int Wd, Ht;
unsigned char Palette[768];
extern int nn[3];
extern int ntot;
extern double tol[5];



int ReadImage(FILE *Infp,IMAGE *Image,int *ImWd,int *ImHt,int *BytePerPixel);
void DisplayIm(IMAGE Im, int ImWd, int ImHt, int ImType);
void PrintIm(IMAGE Im, int ImWd, int ImHt, int ImType);
void TransformIm(IMAGE Im, int ImWd, int ImHt, int ImType);
void CoreIm( double *ImTran, int m);
void Restore(IMAGE Im, double *ImTran, long int Shift, int ImWd, int ImHt);
void Im_ImTran(IMAGE Im, double *ImTran, long int Shift, int ImWd, int ImHt);
void Wavelets(double *x, int nn[], int ndim, int num);
void CutImTran( double *ImTran, int ImWd, int ImHt);
```

```
extern void wtn(double *,int [],int ,int );
extern void ArrSort(entry *arr);
extern int cmpentry(const void *, const void *);

int part; // percentage of coefficients taken
int main(int argc, char *argv[])
{
        unsigned long I, J, Offset, Offset2;
        int b, Err;
        int i;

        part = atoi(argv[2]);

        if(argc !=3) {
        printf("imtest <pcx_image_file>\n");
                exit(1);
        }
        Infp = fopen(argv[1], "rb");
        if (Infp == NULL) {
                printf("Could not open %s.\n", argv[1]);
                exit(1);
        }
        if ((Err=ReadImage(Infp, &Im1, &Wd, &Ht, &b)) < 0) {
               printf("Failed to read the PCX image: %s.\n", argv[1]);
```

```
        printf("Error code: %d\n", Err);
        exit(1);
}

//If the image is a 24-bit color image, convert it to a
//grayscale version
//and leave the end of the image array unused.
if (b == 3) {
        ColorToGray(Im1, Im1, Wd, Ht);
        b = 1;
}

if (SetupDisplay(M320x200x256 | COLORPALETTE)) {
  // Resize the image so that it is now 128 by 128 pixels.
  //If the image is the same size, the following steps
  //effectively copy the image in Im1 to the Im array.
  Im = (IMAGE)malloc(128L*128L);

  if (Im == NULL) {
        printf("Out of memory. \n");
        EndDisplay();
        exit(1);
  }
```

```
ResizeIm(Im1, Wd, Ht, Im1, 128, 128);
//Copy the resized image to the Im array
for (J=0; J<128; J++) {
              Offset = J * Wd;
              Offset2 = J * 128;
              for (I=0; I<128; I++) {
                        Im[Offset2+I] = Im1[Offset+I];
              }
}
free(Im1);   //Free the original image's memory
Wd = 128;
Ht = 128;
b = 1;
DisplayIm(Im, Wd, Ht, b); //Display the new image
getch();
PrintIm(Im, Wd, Ht, b);
TransformIm(Im,Wd,Ht,b); //Direct & inverse transform
PrintIm(Im, Wd, Ht, b);
DisplayIm(Im,Wd,Ht,b);
EndDisplay();
////////////////////////////////
// Make a grayscale palette
for (i = 0; i < 256; i++){
    Palette[i*3] = i;
```

```
            Palette[i*3+1] = i;
            Palette[i*3+2] = i;
          }
          Outfp = fopen("mygirl.pcx", "wb");
          if (Outfp == NULL) {
            printf("Could not open output file: TEST.PCX \n");
            exit(1);
          }
          if(!WritePCXImage(Outfp, Im, Wd, Ht, b, Palette)){
            printf("Write operation not successful \n");
            exit(1);
          }
          fclose(Outfp);
          ////////////////////////////////////
        }
        fclose(Infp);
        farfree(Im);
        return 0;
}

int ReadImage(FILE *Infp, IMAGE *Image, int *ImWd, int *ImHt,
                        int *BytesPerPixel)
//Reads the image from the specified PCX file
```

```
{
        PCXHEADER PCXHdr;
        unsigned char *Row;
        IMAGE  Im;
        unsigned long I, J, ImSize;

        if (!ReadPCXHeader(Infp, &PCXHdr))   //Read the header
                return -1;
        if (!IsValidPCX(&PCXHdr))            //Does the header say the
                return -2;          //image is the right type ?

        //Extract the image dimentions from the header
        GetPCXInfo(&PCXHdr, ImWd, ImHt, BytesPerPixel);
        ImSize = (unsigned long) (*ImWd)*
             (unsigned long)(*ImHt) * (unsigned long)(*BytesPerPixel);
        Im = (IMAGE)farmalloc(ImSize);
        if (Im == NULL)
                return -3;          //Not enough memory

        //Allocate memory for one image row
        Row = (unsigned char *)malloc((*ImWd)*(*BytesPerPixel));
        if (Row == NULL)
                return -4;          //Not enough memory
        for( J=0; J<*ImHt; J++) {  // Read each image row
```

```
        if (!GetPCXRow(Infp, Row, PCXHdr.NPlanes,
                PCXHdr.BytesPerRow, *ImWd)) {
                free(Row);
                return -5;          //Could not read the image
        }

        //Copy the row to the image
        for (I=0; I<*ImWd*(*BytesPerPixel); I++)
                Im[J*(*ImWd)*(long)(*BytesPerPixel)+I] = Row[I];

         }
         free(Row);
         *Image = Im;
         return 1;
}


void DisplayIm(IMAGE Im, int ImWd, int ImHt, int ImType)
//Displays an image of the size ImWd by ImHt at the top-right of the screen.
//The image is clipped to the dimensions of the screen.

{
         unsigned long I, J, Offset;
```

```
        for (J=0; J<ImHt && J<SCREENHT; J++) {
            Offset = J * ImWd;
            for (I=0; I<ImWd && I<SCREENWD; I++) {
                if (ImType == 3)
                   PutPixel(0, I, J,
                   Im[(Offset+I)*3],Im[(Offset+I)*3+1],Im[(Offset+I)*3+2]);
                else
                   PutPixel(0, I, J,
                   Im[Offset+I], Im[Offset+I],Im[Offset+I]);
            }
        }

}


void PrintIm(IMAGE Im, int ImWd, int ImHt, int ImType)
//Print array Im in decimal form to file <pixel.dat>
{
         unsigned long I, J, Offset;
         FILE *gp;

         gp = fopen("pixel.dat","w");
         for (J=0; J<ImHt && J<SCREENHT; J++) {
            Offset = J * ImWd;
```

```
            for (I=0; I<ImWd && I<SCREENWD; I++) {
                if (ImType == 3)
                   fprintf(gp, "%u %u %u ",
                   Im[(Offset+I)*3],Im[(Offset+I)*3+1],Im[(Offset+I)*3+2]);
                else
                   fprintf(gp,"%.2lf ", (double)Im[Offset+I]);
                }
                fprintf(gp,"\n");
        }
        fclose( gp );
}


int cmpentry(const void *a, const void *b)
// cmpentry halps qsort sort in descing order

{
  double x, y;
  x = fabs(((entry *)a)->p);
  y = fabs(((entry *)b)->p);

  if (x < y)
                return 1;
       if (x == y)
```

```
                return 0;
        if (x > y)
                return -1;
}



void TransformIm(IMAGE Im, int ImWd, int ImHt, int ImType)
//Transform image using wtn
//The image is clipped to the dimensions of the screen.
// m = 1 direct transform, m = -1 inverse transform
// Im divides to four quaters with indexes
// 1st quarter from 0           to 64 * 64 - 1
// 2nd quarter from 64 * 64     to 2 * 64 * 64 - 1
// 3rd quarter from 2 * 64 * 64 to 3 * 64 * 64 - 1
// 4th quarter from 3 * 64 * 64 to 4 * 64 * 64 - 1
{

          unsigned long int Shift;
          int m;
          double *ImTran;


          //PrintIm(Im, Wd, Ht, ImType);
```

```
ImWd = 64;   //maximum dimention allowed for dynamic memory
ImHt = 64;   //alocation

nn[0]= ImHt;  // we have two dimentional transformation
nn[1]= ImWd;  // with sizes 64*64. So the theard dimention set to 1
nn[2]= 1;
ntot = nn[0]*nn[1]*nn[2];
tol[0]=0.01;

ImTran = (double  *)calloc(ImWd*ImHt,sizeof(double));
if(ImTran==NULL)
       {printf("no enough memory for ImTran\n");exit(1);}


// Direct transform
m = 1;
Shift = 0;       // 1st quarter of array Im
Im_ImTran(Im, ImTran, Shift, ImWd, ImHt);
CoreIm( ImTran,  m);
CutImTran( ImTran, ImWd, ImHt);
// Inverse transform
m = -1;
CoreIm( ImTran, m);
```

```
// Restore the 1st quarter of array Im
Restore(Im, ImTran, Shift, ImWd, ImHt);

// Direct transform
m = 1;
Shift = 64*64;   // 2nd quarter of array Im
Im_ImTran(Im, ImTran, Shift, ImWd, ImHt);
CoreIm(ImTran, m);
CutImTran( ImTran, ImWd, ImHt);
// Inverse transform
m = -1;
CoreIm( ImTran, m);
// Restore the 2nd quarter of array Im
Restore(Im, ImTran, Shift, ImWd, ImHt);


// Direct transform

m = 1;
Shift = 2*64*64; // 3rd quarter of array Im
Im_ImTran(Im, ImTran, Shift, ImWd, ImHt);
CoreIm(ImTran, m);
CutImTran( ImTran, ImWd, ImHt);
// Inverse transform
```

```
        m = -1;
        CoreIm( ImTran, m);
        // Restore the 3rd quarter of array Im
        Restore(Im, ImTran, Shift, ImWd, ImHt);


        // Direct transform
        m = 1;
        Shift = 3*64*64; // 4th quarter of array Im
        Im_ImTran(Im, ImTran, Shift, ImWd, ImHt);
        CoreIm( ImTran, m);
        CutImTran( ImTran, ImWd, ImHt);
        // Inverse transform
        m = -1;
        CoreIm( ImTran, m);
        //Restore the 4th quarter of array Im
        Restore(Im, ImTran, Shift, ImWd, ImHt);

        free(ImTran);

}//end  TransformIm

void Im_ImTran(IMAGE Im, double *ImTran, long int Shift,
               int ImWd, int ImHt)
```

```
// copy quarter of array Im to array ImTran
{
        unsigned long I, J, Offset;


        for (J=0; J<ImHt && J<SCREENHT; J++) {
                Offset = J * ImWd;
                for (I=0; I<ImWd && I<SCREENWD; I++) {
                    ImTran[Offset+I]= (double)Im[Shift + Offset + I] ;
                }
        }
}// end Im_ImTran



void Restore(IMAGE Im, double *ImTran, long int Shift,
             int ImWd, int ImHt)
//Restore one quarter of array Im
{
        unsigned long I, J, Offset;


        for (J=0; J<ImHt && J<SCREENHT; J++) {
                Offset = J * ImWd;
```

```
                for (I=0; I<ImWd && I<SCREENWD; I++) {
                    Im[Shift + Offset + I] =ROUND(ImTran[Offset + I]);
                }
        }
}/*end Restore */



void CutImTran( double *ImTran, int ImWd, int ImHt)
// Cuting  array ImTran of Descrite Wavelet Transform  coefficients with
// size ImWd * ImHt of non zero elements to size of "part"
// non zero elements
{
        long int Portion;
        entry *arr;
        int i, j;

        Portion = ImWd * ImHt * (double)part/100;
        arr = (entry  *)calloc(ImWd*ImHt,sizeof(entry));
        if(arr == NULL )
               {printf("not enough memory for arr \n");exit(1);}

        ntot = ImWd * ImHt * 1; // 1 corresponds the third dimention
        for (i = 0; i < ntot; i++) {
```

```
                arr[i].p = ImTran[i];
                arr[i].index = i;
        }
        qsort((void *)arr, (size_t)ntot, (size_t)sizeof(entry), cmpentry);

        for (i=Portion; i < ntot; i++){
            arr[i].p = 0.0;
        }
        ArrSort(arr);
        for (i=0; i < ntot; i++) {
                j = arr[i].index;
                ImTran[j] = arr[i].p;
        }
        free(arr);
}//end CutImTran


void CoreIm( double *ImTran, int m)
//Take array ImTran with size 64*64 and make Disrete Wavelet Transform
// m = 1 direct transform
// m = -1 inverse transform
{
        wtn(ImTran,nn,3,m);
}//end CoreIm
```

```
void message(char *msg)
{
  FILE *fp;
        fp = fopen("debug.txt","w");
        fprintf(fp,"%s",msg);
        fclose(fp);
}
```

```
//GRPHICS.C: Various functions used to support DOS and Windows graphics
//programs.
#if !defined(FORWINDOWS)
#include <dos.h>
#include <stdlib.h>
#endif
#include "grphics.h"

int ScrnMode;                    //The current screen mode
int ScrnWd, ScrnHt;              //The current screen's width and height
unsigned int MaxColor;           //Maximum color index available
unsigned int ScrnSeg=0;          //Current video segment
unsigned char UseColorPalette;   //Nonzero if palette holds colors
int NextPaletteNdx;              //Next free location in color palette
float AspectRatio = 1.0;         //Aspect ratio of screen
struct MODE {              //Contains information about a video mode
   int ModeVal;            //Mode register value
   int Wd;                 //Width of mode
   int Ht;                 //Height of mode
   float AspectRatio;      //Aspect ratio of mode
};
struct MODE Modes[5] = {
   {0x03, 0, 0, 1},        //Text mode, 80x25 color
   {0x13, 320, 200, 1.33},// 320x200x256; 256 color
```

```
   {0x2E, 640, 480, 1},    // 640x480x256
   {0x2E, 640, 480, 1},    //Tseng 640x480 32K colors
   {0x03, 800, 600, 1.1}  //Tseng 800x600 32K colors
};
#if !defined(FORWINDOWS)
//A logical palette that is used to add palette entries when
//true 256-color images are displayed
struct PALETTECOLOR{          //A palette entry has red, green,
unsigned r, g, b;             //and blue components
};
struct PALETTECOLOR LogPalette[256];  //A logical color palette
#endif

int SetupDisplay(int Mode)
//In a DOS application, initialize the praphics system
{
#if  !defined(FORWINDOWS)
   union REGS Regs;
   ScrnMode =   Mode & 0x7F;
   ScrnWd =   Modes[ScrnMode].Wd;
   ScrnHt = Modes[ScrnMode].Ht;
   ScrnSeg =  0;
   UseColorPalette =0;
   NextPaletteNdx = 0;
```

```
switch(ScrnMode){
     case MTEXT:
     case M320x200x256:
     case M640x480x256:
             Regs.x.ax = Modes[ScrnMode].ModeVal;// AH = 0x00 AL = mode
             int86(0x10, &Regs, &Regs);
             MaxColor = (ScrnMode == MTEXT) ? 0 : 255;
             if(Mode & COLORPALETTE){
             UseColorPalette = 1;
             SetVGAPalette(0,0,0);  //Set the background color black
             }
             else if (Mode & MTEXT) return 0; //addition
             else                          // Use a gray scale palette
                     SetVGA64Palette();// Create gray scale palette
             break;
     case M640x480x32K:
     case M800x600x32K:
             //Set AX to 0x10F0 to select the Hicolor mode
             Regs.x.ax = 0x10F0;          // AH = 0x10 AL = 0xF0
             Regs.x.bx = Modes[ScrnMode].ModeVal;
             int86(0x10, &Regs, &Regs);
             MaxColor = 32767;     // 32K modes don't have a palette
             break;
     default:
```

```
            return 0;                    // Return failure flag
    }
    AspectRatio = Modes[ScrnMode].AspectRatio;
#endif              // Don't do anything if compiling for Windows
    return 1;
}



void EndDisplay(void)
//In a DOS application, wait for a keypress and then exit graphics mode
{
#if  defined(FORWINDOWS)
#else
        getch();            //Wait for a keypress
        SetupDisplay(MTEXT);//Set the display to color text mode
#endif
}

void PutPixel(int hDC, int X, int Y,
        unsigned char Red, unsigned char Green, unsigned char Blue)
//Writes a pixel to the screen. Supports 32K-color mode for Tseng Labs
//ET-4000 chipset equipped with a Sierra Hicolor DAC.
{
```

```
#if defined(FORWINDOWS)
        SetPixel(hDC, X, Y, PALETTERGB(Red,Green,Blue));
#else
unsigned long Pixel;
unsigned char WriteSeg;
unsigned int far *ScreenPtr;
unsigned char far *VGAScreenPtr;
unsigned int Color;

if(MaxColor == 255){
  // Convert RGB value to a black and white value. The 12 comes from
  // the fact that we are taking the average of three values and only
  //six bits of each color component are used. Thus, they are also
  // divided by four.
        if(UseColorPalette)   // Add color to palette if there is a room
        Color = SetPaletteColor(Red,Green,Blue);
else                    // use a grayscale value
        Color = (Red+Green+Blue)/12 + 64;
        Pixel = X + (long)Y*ScrnWd;
        if(ScrnMode == M640x480x256){
                WriteSeg = Pixel /65536L;  // There are 64K pixels/segment
                outp(0x3CD,WriteSeg);      // Set the segment register
        }
        FP_SEG(VGAScreenPtr) = 0XA000;
```

```
        FP_OFF(VGAScreenPtr) = (unsigned int)(Pixel & 0xFFFF);
        *VGAScreenPtr = Color;                  // set the pixel's color
        }
        else {                                  // 32k colors
          Color = ((Red/8 & 31) <<10) | ((Green/8 & 31) <<5) | (Blue/8 &31);
          // Compute the pixel location in memory. Note: There are
          // two bytes per pixel.
          Pixel = (X+Y*(unsigned long)ScrnWd)*2;
          // Determine which 64K segment the pixel is in
          WriteSeg = Pixel /65536L;
          if (WriteSeg != ScrnSeg){
             outp(0x3CD, WriteSeg); // Set the segment register
             ScrnSeg = WriteSeg;    // Remember the segment
          }
          FP_SEG(ScreenPtr) = 0XA0000;
          FP_OFF(ScreenPtr) = (unsigned int) (Pixel & 0xFFFF);
          *ScreenPtr = Color;    // Set the pixel's color
        }
#endif
}
#if !defined(FORWINDOWS)
#define WRITE_PIXEL 0x3c8
void SetVGA64Palette(void)
{
```

```
  int i;
  (void)outp(WRITE_PIXEL, 64);
  for(i=0; i<64; i++){
        (void)outp(WRITE_PIXEL+1, i);
        (void)outp(WRITE_PIXEL+1, i);
        (void)outp(WRITE_PIXEL+1, i);
  }
}

void SetVGAPalette(unsigned char Red, unsigned char Blue,
        unsigned char Green)
{
   (void)outp(WRITE_PIXEL, NextPaletteNdx);
   (void)outp(WRITE_PIXEL+1, Red/4);
   (void)outp(WRITE_PIXEL+1, Green/4);
   (void)outp(WRITE_PIXEL+1, Blue/4);
   LogPalette[NextPaletteNdx].r = Red;
   LogPalette[NextPaletteNdx].g = Green;
   LogPalette[NextPaletteNdx].b = Blue;
   NextPaletteNdx++;
}

unsigned int  SetPaletteColor(unsigned char Red, unsigned char Green,
        unsigned char Blue)
```

```
{
// If the color does not already exist in the palette, then add it in if
// there is room. If there isn't any more room in the palette, then use
// the closest matching color.
   int i, MinNdx;
   unsigned long MinDist, Dist;
   unsigned int DistR, DistG, DistB;

   // See if color is already in the logical palette
   for(i=0;i<NextPaletteNdx; i++)
   // If it is, use its palette color
   if(LogPalette[i].r == Red && LogPalette[i].g ==
      Green && LogPalette[i].b == Blue)
        return i;
   // The color isn't in the palette. Is there room to add it?
   if(NextPaletteNdx >= 256){
   i=0;                    // If there isn't room for a
   MinNdx = 0;             // new color, use the closest
   MinDist = 0xffff;       // matching color
   for(i=0; i<256; i++){
        DistR = abs(LogPalette[i].r - Red);
        DistG = abs(LogPalette[i].g - Green);
        DistB = abs(LogPalette[i].b - Blue);
        Dist = (DistR + DistG + DistB) /3;
```

```
            if(MinDist > Dist){
            MinDist = Dist;
            MinNdx = i;
            }
        }
        return MinNdx;
        }
        else {
            SetVGAPalette(Red, Green, Blue);
            // Return the index of the color just added to the palette
            return NextPaletteNdx -1;
            }
}
#endif
```

```
//IMTOOLS.C: A collection of image processing tools.

#include <math.h>
#include <stdlib.h>
#include "grphics.h"
#include "imtools.h"

void ResizeIm(IMAGE InIm, int InWd, int InHt, IMAGE OutIm,
              int OutWd, int OutHt)
//Copies and resizes the image in InIm, which is ImWd by ImHt in size, to
//the OutIm array, which is OutWt by OutHt in size.

{
    unsigned long I;

    for (I=0; I<InHt;I++)  //Stretches image horizontally
        ZoomIm(&InIm[I*InWd],InWd,1,&OutIm[I*InWd], OutWd, 1);
    for (I=0; I<OutWd;I++) //Stretches image vertically
        ZoomIm(&OutIm[I],InHt,InWd, &OutIm[I],OutHt, InWd);
}



void ZoomIm(IMAGE InIm, int Currlen, int InStep, IMAGE OutIm, int NewLen,
```

```
            int OutStep)
//Zooms in on a portion of an image in one dimention. Currlen is the current
//dimension's length. NewLen is to be the new length. The ratio of these two
//values determines how much the input image is streched or compressed.
//InStep and OutStep are set to 1 to stretch or compress the image
//horizontally, set InStep to the width of the input image and OutStep to the pixe
//width of the output image.

{
    unsigned long I=0, Pixel =0;
        float Accum = 0, Val, Scale, InAmt=1.0, UseAmt;

        Scale = (double)NewLen / Currlen;
        UseAmt = 1/ Scale;
        //Compute the output pixels
        while (Pixel < (unsigned long) NewLen * OutStep) {
                //Linearly interpolate the current image pixel
                Val = InAmt * InIm[I] + (1.0-InAmt) * InIm[I + InStep];
                //Use what's in InAmt. However, there still isn't enough
                //to make up a whole output pixel.
                if (InAmt < UseAmt) {
                        Accum += Val * InAmt; //Add in weighted contribution
                        UseAmt -= InAmt;
                        InAmt = 1.0;
```

```
                I += InStep;
            }
            else {
            //Input pixel is not completely consumed in current pixel
                Accum += Val * UseAmt;
                OutIm[Pixel] = Accum * Scale;
                Pixel += OutStep;
                Accum = 0;
                InAmt -= UseAmt;
                UseAmt = 1/ Scale;
            }
        }
}


void Colorize(IMAGE InIm, IMAGE OutIm, int ImWd,
        int ImHt, float Red, float Green, float Blue)
//Converts a grayscale image (InIm) to a color image by multiplying each
//pixel times the red, green, and blue components passed in. The value of
//the color components should be between 0 and 1.

{
```

```
        unsigned long I, J, Offset;

        for (J=0; J<ImHt;J++) {
            Offset = J * ImWd;
            for (I=0; I < ImWd; I++) {
                OutIm[(I+Offset)*3] = InIm[I+Offset] * Red;
                OutIm[(I+ Offset)*3 +1] = InIm[I+Offset] * Green;
                OutIm[(I+ Offset)*3 +2] = InIm[I+Offset] * Blue;
            }
        }
}

void ColorToGray(IMAGE InIm, IMAGE OutIm, int ImWd, int ImHt)
//Creates a grayscale image from a 24-bit RGB image
{
  unsigned long I, J, Offset;

  for(J=0;J<ImHt;J++){
        Offset = J*ImWd;
        for(I=0;I<ImWd;I++){
        OutIm[I+Offset]=((unsigned int)InIm[(I+Offset)*3]+
                (unsigned int)InIm[(I+Offset)*3+1] +
                (unsigned int)InIm[(I+Offset)*3+2])/3;
                }
```

```
        }
}

void MirrorXIm(IMAGE Im, int ImWd, int ImHt)
//Flips a grayscale image about its center row. Note: this function
//otherwrites the original image with the flipped image.

{
        unsigned long I, J, K, Offset;
        unsigned char Tmp;

        for (J=0; J<ImHt; J++) {
                Offset = J * ImWd;
                for (I=0, K=ImWd-1; I<ImWd/2;I++,K--) {
                        Tmp = Im[Offset+K];
                        Im[Offset+K] = Im[Offset+I];
                        Im[Offset+I] = Tmp;
                }
        }
}


void MirrorYIm(IMAGE Im, int ImWd, int ImHt)
//Flips a grayscale image about its center column. Note: this function
```

```
//overwrites the input image with the flipped image.
{
        unsigned long I, J, K, Offset, Offset2;
        unsigned char Tmp;

        for (J=0, K=ImHt-1; J<ImHt/2;J++,K--){
                Offset = J * ImWd;
                Offset2 = K * ImWd;
                for (I=0; I<ImWd; I++) {
                        Tmp = Im[Offset2+I];
                        Im[Offset2+I] = Im[Offset+I];
                        Im[Offset+I]= Tmp;
                }
        }
}



void MirrorXYIm(IMAGE Im, int ImWd, int ImHt)
//Flips a grayscale image about its diagonal, which effectively
//rotates the image 90 degrees counterclochwise. Note: this function
//overwrites the original image with the flipped image.

{
```

```
        unsigned long I, J, C, Offset, Offset2;
        unsigned char Tmp;

        for (J=0; J<ImHt; J++) {
                Offset = J * ImWd;
                for (I=0; I<J; I++) {
                        Offset2 = I * ImWd + J;
                        Tmp = Im[Offset2];
                        Im[Offset2] = Im[Offset+I];
                              Im[Offset+I] = Tmp;
                }
        }
}


int Average(IMAGE InIm, IMAGE OutIm, int ImWd, int ImHt)
//Averages 3 by 3 neighborhoods of pixels in a grayscale image. The function
//allocates one row of temporary storage. If this allocation fails, the
//function returns 0. Otherwise, it returns 1. The borders of the image are
//not changed.
{
          IMAGE PrevRow;
```

```
unsigned long I, J, Offset, Pix;
int Ave, PrevCol;

//Allocate memory to hold the last row of the image processed
PrevRow = (unsigned char *)malloc(ImWd);
if (!PrevRow)
                return 0;    //Not enough memory
//Copy the top row of the image to the PrevRow array
for (I=0; I<ImWd; I++) {
                PrevRow[I] = (unsigned int)InIm[I];
                OutIm[I] = InIm[I];
                OutIm[(ImHt-1)*ImWd+I] = InIm[(ImHt-1)*ImWd+I];
}
for (I=0; I<ImHt; I++) {
                Offset = I * ImWd;
                OutIm[Offset] = InIm[Offset];
                OutIm[Offset+ImWd-1] = InIm[Offset+ImWd-1];
}
//Edges of the image are left unchanged
for (J=1; J<ImHt-1; J++) {
                Offset = J * ImWd;
                PrevCol = (unsigned int)InIm[Offset];
                for (I=1; I<ImWd-1; I++) {
                    Pix = Offset +I;
```

```
                    Ave = ((int)PrevRow[I-1] + (int)PrevRow[I] +
                    (int)PrevRow[I+1] + PrevCol + (int)InIm[Pix]+
                    (int)InIm[Pix+ImWd] + (int)InIm[Pix+ImWd+1])/9;
                    PrevCol = InIm[Pix];
                    PrevRow[I] = InIm[Pix];
                    OutIm[Pix] = (unsigned char)Ave;
                }
                PrevRow[0] = InIm[Offset];
                PrevRow[ImWd-1] = InIm[Offset+ImWd-1];
        }
        free(PrevRow);
        return 1;
}

void Histogram(unsigned int *Hist, IMAGE InIm, int ImWd, int ImHt)
//Returns the histogram of a grayscale image. You must allocate space
//for the histogram array before calling the routine. Therefore, to allocate
//a histogram array large enough to hold 256 gray levels, use:
//hist = (unsigned int *)malloc(NUMBEROFGRAYLEVELS*sizeof(int));

{
        unsigned long I, J, Offset;

        for (I=0; I<256; I++)
```

```
                Hist[I] = 0;
        for (J=0; J<ImHt; J++) {
                Offset = J * ImWd;
                for (I=0; I<ImWd; I++)
                                Hist[InIm[I+Offset]]++;
        }
}


int  ContrastEnhance(IMAGE InIm, IMAGE OutIm, int ImWd, int ImHt,
                        int LowVal, int HiVal)
//Stretches the histogram of the image in order to improve the contrast.
//This function cuts out all pixels below LowVal and above HiVal and
//then stretches the remaining pixels in the histogram so that they
//are evenly distributed across the image. This function currently only
//works with grayscale images.

{
        unsigned long I, J, Offset, Pix;
        float Scale;


        Scale = 255.0 / (float)(HiVal - LowVal);
        for (J=0; J<ImHt; J++) {
```

```
                    Offset = J * ImWd;
                    for (I=0; I<ImWd;I++){
                        Pix = Offset + I;
                        if (InIm[Pix] < LowVal)
                            OutIm[Pix] = 0;
                        else if (InIm[Pix] > HiVal)
                            OutIm[Pix] = 255;
                        else
                            OutIm[Pix] =(unsigned char)(Scale *(float)
                                        (InIm[Pix]-LowVal));
                    }
        }
        return 1;
}

void BrighenIm(IMAGE InIm, IMAGE OutIm, int ImWd, int ImHt, int Val)
//Adds a value to each pixel in a grayscale image
{
        unsigned long I, J, Offset;
        unsigned int Tmp;

        for (J=0; J<ImHt; J++) {
            Offset = J * ImWd;
            for (I=0; I<ImWd; I++) {
```

```
                Tmp = (unsigned int)InIm[Offset+I] + Val;
                OutIm[Offset+I] = (Tmp > 255) ? 255: (unsigned char)Tmp;
            }
        }
}


void GammaCorr(IMAGE InIm, IMAGE OutIm, int ImWd, int ImHt, double Val)
//Applies a gamma correction to an RGB image. Set Val between 0 and 1
//to lighen the image and greater than 1 to darken it. To gamma correct
//for an image that looks good on a monitor, a good value to use is 0.45.
//The function assumes that the image is an RGB image.
{
        unsigned long I, J, Offset, Offset2;
        double Res;

        for (J=0; J < ImHt; J++) {
            Offset = J * ImWd;
            for (I=0; I<ImWd; I++) {
                Offset2 = (Offset + I) * 3;
                Res = pow((double)InIm[Offset2], Val);
                OutIm[Offset2] = (Res > 255) ? 255 : (unsigned char)Res;
                Offset2++;
```

```
                Res = pow((double)InIm[Offset2], Val);
                OutIm[Offset2] = (Res > 255) ? 255 : (unsigned char)Res;
                Offset2++;
                Res = pow((double)InIm[Offset2], Val);
                OutIm[Offset2] = (Res > 255) ? 255 : (unsigned char)Res;
            }
        }
}
```

```
// PCXTOOL.C: PCX toolkit that supports 256-color and 24-bit color images.
#include <stdio.h>
#include "pcxtool.h"
void PrintPCXHeader(PCXHEADER *);

int WritePCXHeader(FILE *Outfp, PCXHEADER *PCXHdr)
// Write a PCX header to a file. The header is written before the image is
// placed in the file. Returns 0 if the write operation fails. Otherwise,
// it reurns 1. The header should be initialized to its proper settings
// before calling this routine.

{
  if(fseek(Outfp, 0L, SEEK_SET)) return 0;
  // Write header to file
  if(!fwrite(PCXHdr, sizeof(PCXHEADER),1,Outfp)) return 0;
  return 1;
}

void SetPCXHeader(PCXHEADER *PCXHdr, int ImWd, int ImHt, int BytesPerPixel)
// Set the default values in a PCX header. ImWd and ImHt are the image's
// pixel width and height. Set BytesPerPixel to 1 for 256-color images
// and 3 for RGB images.
{
    PCXHdr->PCXId = 0x0A;
```

```
    PCXHdr->Version = 5;
    PCXHdr->Encoding = 1;           // Use run-length encoding
    PCXHdr->BitsPerPixel = 8;       // 8 bits per pixel
    PCXHdr->LeftX = 0;              // Display image at (0,0)
    PCXHdr->LeftY = 0;
    PCXHdr->RightX = ImWd -1;
    PCXHdr->RightY = ImHt -1;
    PCXHdr->DisplayXRes = 640;      // Screen resolution
    PCXHdr->DisplayYRes = 480;
    *PCXHdr->Palette = '\0';
    PCXHdr->NPlanes = BytesPerPixel; // Number of planes equals the
    PCXHdr->BytesPerRow = ImWd;      // number of bytes per pixel
    // RGB images don't have a palette, but 256-color images do
    PCXHdr->PaletteInfo = (BytesPerPixel == 1) ? 1 : 0;
}

int EncodePut(FILE *Outfp, unsigned char Val, unsigned char Count)
// Encodes and writes the value in Val to the PCX file. Count is the
// repeat count for the value.
{
   if(Count == 1 && 0xC0 != (0xC0 & Val)) {
   //Write the value because it only has a length of one
     if(EOF == putc((int)Val, Outfp))
       return 0;   // Disk write error
```

```
  }
  else {
   // The value is greater than 0xC0 or it represents a repeat count.
   // Write them as two bytes in the file.
   // First write the run-length and then the value.
   if(EOF == putc((int)(0xC0 | Count), Outfp))
       return 0;              // Disk write error
   if(EOF == putc((int)Val, Outfp))
       return 0;              // Disk write error
  }
  return 1;
}

int WritePCXImage(FILE *Outfp, unsigned char *Image, int ImWd,
        int ImHt, int BytesPerPixel, unsigned char *Palette)
// Writes the image data to a PCX file. Rhis code handles 256-color
// and 24-bit color images. Returns 1 if successful;
// 0 otherwise. Set Palette to NULL and BytesPerPixel to 3 for
// 24-bit images. Note: This function only handles images less than 64K.

{
   int y;
   PCXHEADER PCXHeader;
   SetPCXHeader(&PCXHeader, ImWd, ImHt, BytesPerPixel);
```

```
   if(!WritePCXHeader(Outfp,&PCXHeader)) return 0;
   for(y=0;y<ImHt;y++){
   if(!WritePCXRow(Outfp,&Image[y*ImWd*BytesPerPixel],
        ImWd, BytesPerPixel)) return 0;
   }
   if((Palette != NULL) && (BytesPerPixel == 1))
        if(!WritePCXPalette(Outfp,Palette)) return 0;
   return 1;    // Success
}


int WritePCXRow(FILE *Outfp, unsigned char *Image, int ImWd,
        int BytesPerPixel)
// Write a single line of an image to a PCX file. This routine only
// supports 256-color and 24-bit RGB images. The 256-color images are
// written one byte (pixel) at a time to the file. The RGB images are
// written as a red row, then green row, the blue row. Set bytesPixel
// to 1 for 256-color images and 3 for RGB images.
{
   unsigned char PrevVal, Val, Rep;
   int x, i, p, EndOfRow;
   for(p=0;p<BytesPerPixel; p++){
   x = p;                        // First pixel on image row to write
```

```
PrevVal = Image[x];          // First image pixel
x +=BytesPerPixel;           // Go to the next pixel in the row
EndOfRow = ImWd * BytesPerPixel + p; // Last pixel to write
Rep =1 ;                     // You begin with only one pixel
do {
     Val = Image[x];         // The next image pixel to write
     x += BytesPerPixel;
     if(Val == PrevVal){
     // If the next is the same as the current, increment the
     // repetition counter
     Rep++;
     // Maximum run count reached. Write out current data.
     if(Rep == 63) {
     if(!(i=EncodePut(Outfp,PrevVal,Rep)))
        return 0;            // Write operation failed
     Rep = 0;                // Reset the repetition counter
     }
}
else {                       // PrevVal != Val, so write PrevVal if
     if(Rep) {               // the repetition is not zero
     if(!(i=EncodePut(Outfp,PrevVal,Rep))) return 0;
     }
     PrevVal = Val;          // Set the previous pixel to the current
     Rep = 1;                // one and reset the run-length counter
```

```
        }
    } while (x < EndOfRow); // Reached end of the image row ?

    if(Rep) {                // Write out unwritten pixels on line
    if(!(i=EncodePut(Outfp,PrevVal,Rep)))
       return 0;
    }
    // If there are an odd number of bytes in the image row,
    // write another pixel to the row
    if (ImWd % 2 == 1)
       EncodePut(Outfp, 0, 1);
    }
    return 1;
}


int WritePCXPalette(FILE *Outfp, unsigned char *Palette)
// Writes a 256-color palette to the end of the PCX file. The palette
// should be 256*3 (768) bytes long. It should have 256 RGB triples.
// The palette is written to the end of the file and is preceded by a
// value of 12.

{
    unsigned char Code=12;
```

```
   if(fseek(Outfp, 0L, SEEK_END)) return 0;
   if(!fwrite(&Code, 1, 1, Outfp)) return 0;
   if(!fwrite(Palette, 256*3, 1, Outfp)) return 0;
   return 1;
}

int ReadPCXHeader(FILE *Infp, PCXHEADER *PCXHdr)
{
// Reads a PCX file's header. Returns 0 if the read operation fails.
// Otherwise, it returns 1.
   if(fseek(Infp, 0L, SEEK_SET)) return 0;
   if(fread(PCXHdr, sizeof(PCXHEADER), 1, Infp) != 1)
      return 0;    // Could not read header
   return 1;
}

void GetPCXInfo(PCXHEADER *PCXHdr, int *ImWd, int *ImHt, int *BytesPerPixel)
// retrieves the image width, height, and the number of bytes per color
// from the header PCXHdr
{
   *ImWd = PCXHdr->RightX - PCXHdr->LeftX + 1;
   *ImHt = PCXHdr->RightY - PCXHdr->LeftY + 1;
   *BytesPerPixel = PCXHdr->NPlanes;
}
```

```
int IsValidPCX(PCXHEADER *PCXHdr)
// Returns a value of 1 if the header passed to the routine represents
// either a 256-color or 24-bit image for which PCXTOOL.C is designed
{
  if(PCXHdr->PCXId == 0x0A && PCXHdr->Version == 5) {
      // Correct format but is the image type one supported ?
      if(PCXHdr->Encoding == 1 &&     // Uses run-length encoding
         PCXHdr->BitsPerPixel == 8){  // 8 bits per pixel
         if(PCXHdr->NPlanes == 3)
            return 1;                 // 24-bit RGB image
         else if(PCXHdr->NPlanes == 1)
            return 1;                 // 256-color image
      }
  }
  return 0;
}                          // Image type not supported by PCXTOOL.C



int GetPCXRow(FILE *Infp, unsigned char *Image, int BytesPerPixel,
        unsigned int BytesPerRow, int ImWd)
// Reads and decodes a single pixel row. The image row is returned in the
// Image array. There must be enough room in Image to hold
```

```
// BytesPerRow*BytesPerPixel pixels.
{
   int i, b, x, p, RunLen;

   for(p=0; p<BytesPerPixel; p++) {       // Get each color component
   x = 0;
   while (x < BytesPerRow){               // Get the whole image row
        if(EOF == (b=getc(Infp)))
           return 0;
        if(b > 0xC0) {                    // This is a repeat count
        RunLen = b & 0x3F;
        if(EOF == (b=getc(Infp)))         // Get the pixel value
           return 0;
        }
        else
           RunLen = 1;                     // A single pixel value
        if(x>=ImWd) break;                 // Only copy the image's pixels
        for(i=0; i<RunLen; i++){           // Copy the value to the image
            Image[x*BytesPerPixel + p] = b;
            x++;
        }
      }
   }
   return 1;
```

```
}

int ReadPCXImage(FILE *Infp, unsigned char *Image, int *ImWd,
        int *ImHt, int *BytesPerPixel, unsigned char *Palette)
// sample routine that reads the image data to a PCX file. This
// code handles 256-color and 24-bit color images. Returns 1 if
// successful; 0 otherwise. Set Palette to NULL for 24-bit images.
// The image must be less than 64K in size and the image array
// must be large enough to hold the whole uncompressed image.

{

  int y;
  PCXHEADER PCXHdr;

  if(!ReadPCXHeader(Infp,&PCXHdr))     // Read the headers
     return 0;
  if(!IsValidPCX(&PCXHdr))             // Does the header say the image is
     return 0;                         // the right type ?
  // Extract the image dimensions from the header
  GetPCXInfo(&PCXHdr, ImWd, ImHt, BytesPerPixel);

  for(y=0; y<*ImHt;y++)    //  Read each image row
        if(!GetPCXRow(Infp, &Image[y*(*ImWd)*(*BytesPerPixel)],
```

```
            PCXHdr.NPlanes,PCXHdr.BytesPerRow, *ImWd))
            return 0;
  // Read the image palette if there is one
  if(*BytesPerPixel == 1 && PCXHdr.PaletteInfo == 1)
        ReadPCXPalette(Infp,Palette);
  return 1;
}


int ReadPCXPalette(FILE *Infp, unsigned char *Palette)
// Reads the 256-color palette from the end of the file. The palette should
// be 256*3 bytes in size. The Palette array should be 768 bytes in size.
{
   unsigned char Code;

   // Seek to the location where the palette marker is. It should be the
   // value 12.
   if(fseek(Infp, 256L*-3 -1, SEEK_END))
      return 0;
   if(fread(&Code, 1,1,Infp) !=1)
      return 0;

   if(Code == 12)    // If the marker exists, read the palette
        if(fread(Palette,256*3,1,Infp) != 1)
```

```
            return 0;
        return 1;
}
void PrintPCXHeader(PCXHEADER *PCXHdr)
{
    printf("PCXId = %d\n",PCXHdr->PCXId);
    printf("Version = %d\n",PCXHdr->Version);
    printf("Encoding = %d\n",PCXHdr->Encoding);
    printf("BitsPerPixel = %d\n",PCXHdr->BitsPerPixel);
    printf("LeftX = %d\n",PCXHdr->LeftX);
    printf("LeftY = %d\n",PCXHdr->LeftY);
    printf("RightX = %d\n",PCXHdr->RightX);
    printf("RightY = %d\n",PCXHdr->RightY);
    printf("XRes = %d\n",PCXHdr->DisplayXRes);
    printf("YRes = %d\n",PCXHdr->DisplayYRes);
    printf("NPlanes = %d\n",PCXHdr->NPlanes);
    printf("BytesPerRow = %d",PCXHdr->BytesPerRow);
}
```

```
//  TRANSF.C Performs DWT in two dimensions.
#include "transf.h"
extern void message(char *);

void wtn(double *a,int nn[],int ndim,int isign)
{
  int        ntot, i, j, ii, jj, idim, nnew;
  int        k, n, nprev, nt;
  double *wksp;
  FILE *ofp;
  int Offset;

  ofp = fopen("outfile", "w+");
  for (i=0; i<nn[0]; i++){
      Offset = nn[0]*i;
      for(j=0;j<nn[1]; j++){
           fprintf(ofp," %.2lf", a[Offset +j]);
      }
      fprintf(ofp,"\n");
  }
  fclose(ofp);
  wksp = (double *) calloc(NMAX,sizeof(double));
  if (wksp == NULL){
       printf("\nwksp = NULL");
```

```
        exit(1);
    }
    ntot=1;
    for(idim = 0;idim < ndim;idim++){
            ntot*= nn[idim];
    }
    nprev=1;
    for(idim=0;idim<ndim;idim++){
            n=nn[idim];
            nnew=n*nprev;
            if (n>4) {
               for(ii=0,i=(ntot-1+nnew)/(nnew);i>0;i--,ii+=(nnew)){
                   for(j=1;j<=nprev;j++){
                      jj=j+ii;
                      for (k=0;k<n;k++){
                          wksp[k]=a[jj-1];
                          jj+= nprev;
                      }
                      if (isign>=0) {
                         nt=n;
                         while (nt>=4) {
                               daub20(wksp,nt,isign);
                               nt/= 2;
                         }
```

```
                }
                else{
                   nt=4;
                   while (nt<=n) {
                         daub20(wksp,nt,isign);
                         nt*= 2;
                   }
                }
                jj=j+ii;
                for (k=0;k<n;k++){
                    a[jj-1]=wksp[k];
                    jj+= nprev;
                }
              }//end for
           }// end for
        }//end if
        nprev=nnew;
  }//end for
  free(wksp);
}// end wtn
```

```
void daub20(double *a, int n, int isign)
{
   double ai,ai1,*wksp;
   double sig = -1.0;
   double c20r[21];
   int i,ii,j,jf,jr,k,n1,ni,nj,nh,nmod;
   int ioff,joff;

   if(n < 4)
      return;
   wksp = (double *)malloc((n+1)*sizeof(double));
   for(k=1; k <= 20;k++){
      c20r[21-k]=sig*c20[k];
      sig = -sig;
   }
   ioff =joff = -(n>>1);
   nmod = 20*n;
   n1=n-1;
   nh = n>>1;
   for(j=1;j<=n;j++)
      wksp[j]=0.0;
   if(isign >= 0) {
     for(ii=1,i=1;i<=n;i+=2,ii++){
        ni=i+nmod+ioff;
```

```
      nj=i+nmod+joff;
      for(k=1;k<=20;k++){
         jf=n1 & (ni+k);
         jr=n1 & (nj+k);
         wksp[ii] +=c20[k]*a[jf+1-1];
         wksp[ii+nh] +=c20r[k]*a[jr+1-1];
      }
   }
}
else{
   for(ii=1,i=1;i<=n;i+=2,ii++){
      ai=a[ii-1];
      ai1=a[ii+nh-1];
      ni=i+nmod+ioff;
      nj=i+nmod+joff;
      for(k=1;k<=20;k++){
         jf=(n1 & (ni+k))+1;
         jr=(n1 & (nj+k))+1;
         wksp[jf] +=c20[k]*ai;
         wksp[jr] +=c20r[k]*ai1;
      }//end for
   }//end for
}//end else
for (j=1;j<=n;j++)
```

```
        a[j-1]=wksp[j];
    free(wksp);
}//end daub20

void set_entry(entry *a, entry *b)
{
  a->p = b->p;
  a->index = b->index;
}

void ArrSort(entry *arr)
//Printing sorted array  of coefficients
{
        FILE *gp;

        int ImHt = 64;
        int ImWd = 64;
        unsigned long i, j, Offset;

        gp = fopen("sort.dat","w");
        for (j=0; j<ImHt;j++){
                Offset = j * ImWd;
                for (i=0; i < ImWd; i++) {
                    fprintf(gp,"%.2lf  ", arr[Offset+i].p);
```

```
                }
                fprintf(gp,"\n");
        }
        fclose(gp);
}
```

```
//GRPHICS.H
#ifndef GRPHICSH
#define GRPHICSH

#if defined(FORWINDOWS)
#include <windows.h>
#else
#include <conio.h>
#endif
#include <stdlib.h>
#ifdef __TURBOC__
#include <alloc.h>
#else
#include <malloc.h>
#include <time.h>
time_t _grphicstime;
#define randomize() srand((unsigned)time(&_grphicstime));
#define farmalloc(bytes) halloc(bytes,1)
#define farfree hfree
#endif

#if defined(FORWINDOWS)
#define DECLARE_HDC
#else
```

```
#define MTEXT           0
#define M320x200x256    1
#define M640x480x256    2
#define M640x480x32K    3
#define M800x600x32K    4
#define COLORPALETTE            0x80
#define GREYSCALEPALETTE        0x00
// This macro is used to ensure correct rounding of integer value.
#define ROUND( a )  (((a) < 0 ) ? (int) ((a) - 0.5) : \
                                    (int) ( (a) + 0.5 ))

extern int ScrnMode;
unsigned int SetPaletteColor(unsigned char, unsigned char, unsigned char);
void SetVGAPalette(unsigned char, unsigned char, unsigned char);
void SetVGA64Palette(void);
#endif

extern int ImWd, InHt;
extern float AspectRatio;
void GenImage(int );
int SetupDisplay(int );
void EndDisplay(void);
void PutPixel(int, int, int, unsigned char, unsigned char, unsigned char);
```

```
void Setup(char far *);
void Cleanup(void);

#endif
```

```
//IMTOOLS.H
#ifndef IMTOOLSH
#define IMTOOLSH

typedef unsigned char huge* IMAGE;

void ResizeIm(IMAGE InIm, int ImWd,
              int ImHt, IMAGE OutIm, int OutWd, int outHt);
void ZoomIm(IMAGE InIm, int CurrLen,
           int InStep, IMAGE OutIm, int NewLen,int OutStep);
int SharpenIm(IMAGE InIm, IMAGE OutIm, int ImWd, int ImHt);
void Colorize(IMAGE InIm, IMAGE OutIm, int ImWd, int ImHt,
              float Red, float Green, float Blue);
void ColorToGray(IMAGE InIm, IMAGE OutIm, int ImWd, int ImHt);
void MirrorXIm(IMAGE Im, int ImWd, int ImHt);
void MirrorYIm(IMAGE Im, int ImWd, int ImHt);
void MirrorXYIm(IMAGE Im, int ImWd, int ImHt);
void BrightenIm(IMAGE InIm, IMAGE OutIm, int ImWd, int ImHt, int Val);
int Average(IMAGE InIm, IMAGE OutIm, int ImWd, int ImHt);
void Histogram(unsigned int *Hist, IMAGE InIm, int ImWd, int ImHt);
int ContrastEnhance(IMAGE InIm, IMAGE OutIm, int ImWd, int ImHt, int LowVal,
                    int HiVal);
void Gammacorr(IMAGE InIm, IMAGE OutIm, int ImWd, int ImHt, double Val);
```

```
#endif
```

```
\\PCXTOOL.H
#ifndef PCXTOOLH
#define PCXTOOLH

typedef struct PCXHDR{
    char PCXId;
    char Version;
    char Encoding;
    char BitsPerPixel;
    unsigned int LeftX, LeftY, RightX, RightY;
    unsigned int DisplayXRes, DisplayYRes;
    char Palette[48];
    char Reserved;
    char NPlanes;
    unsigned int BytesPerRow;
    unsigned int PaletteInfo;
    unsigned char Reserved2[58];
} PCXHEADER;

int WritePCXHeader(FILE *, PCXHEADER *);
void SetPCXHeader(PCXHEADER *, int , int , int );
int EncodePut(FILE *, unsigned char , unsigned char );
int WritePCXImage(FILE *, unsigned char *, int , int , int ,unsigned char *);
int WritePCXRow(FILE *, unsigned char *, int , int );
```

```
int WritePCXPalette(FILE *, unsigned char *);
int ReadPCXHeader(FILE *, PCXHEADER *);
void GetPCXInfo(PCXHEADER *, int *, int *, int *);
int IsValidPCX(PCXHEADER *);
int GetPCXRow(FILE *, unsigned char *, int , unsigned int , int );
int ReadPCXImage(FILE *, unsigned char *, int *,int *,int *,unsigned char *);
int ReadPCXPalette(FILE *, unsigned char *);
#endif
```

```
//TRANSF.H
#ifndef TRANSFH
#define TRANSFH
#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <math.h>
#define NMAX  64*64*1
#define MAXSIZE NMAX
#define ALN2I 1.442695022
#define TINY 1.0e-5
#define M 7
#define NSTACK 100
#define END 0x7E
#define TOL1 0.001
#define TOL2 0.001


typedef struct {
        double p;       /* input vector                          */
        int index;      /* index of element in vector            */
        } entry;

typedef struct {
```

```
        int ni;
        int nj;
        int nk;
        double *q1;
        } PDATA;

typedef struct {
        int wnum;
        entry *war;
        } WDATA;

PDATA p;
WDATA w;


static double c20[21]={
0.0,
0.026670057901,
0.188176800078,
0.527201188932,
0.688459039454,
0.281172343661,
-0.249846424327,
-0.195946274377,
```

```
0.127369340336,
0.093057364604,
-0.071394147166,
-0.029457536822,
0.033212674059,
0.003606553567,
-0.010733175483,
0.001395351747,
0.001992405295,
-0.000685856695,
-0.000116466855,
0.000093588670,
-0.000013264203
};



char filein[80];
char fileout[80];
int nn[3];
int i,j;
int counter;
double error[5];
```

```
int ntot;
double tol[5];


double norm2 (double *, double *,int );
void wavelets(double *, int [], int , int );
void store_wavelets(entry *, int , int );
extern void DWT(double *,int [], int , double , int );
double SNRstat(double *, double *, int );
void wtran(int );
void wt1(double *, int , int );
double test(double *, int , int , int );
void  pad(double *, double *, int [], int []);
void set_entry(entry *, entry *);
void daub20(double *, int , int );
double evaluate(double *, double *, int [], int , int , int );
int index(int );
void unpad(double *x, double *c, int nn[], int ni, int nj, int nk);
#endif
```

Clair at 100 %

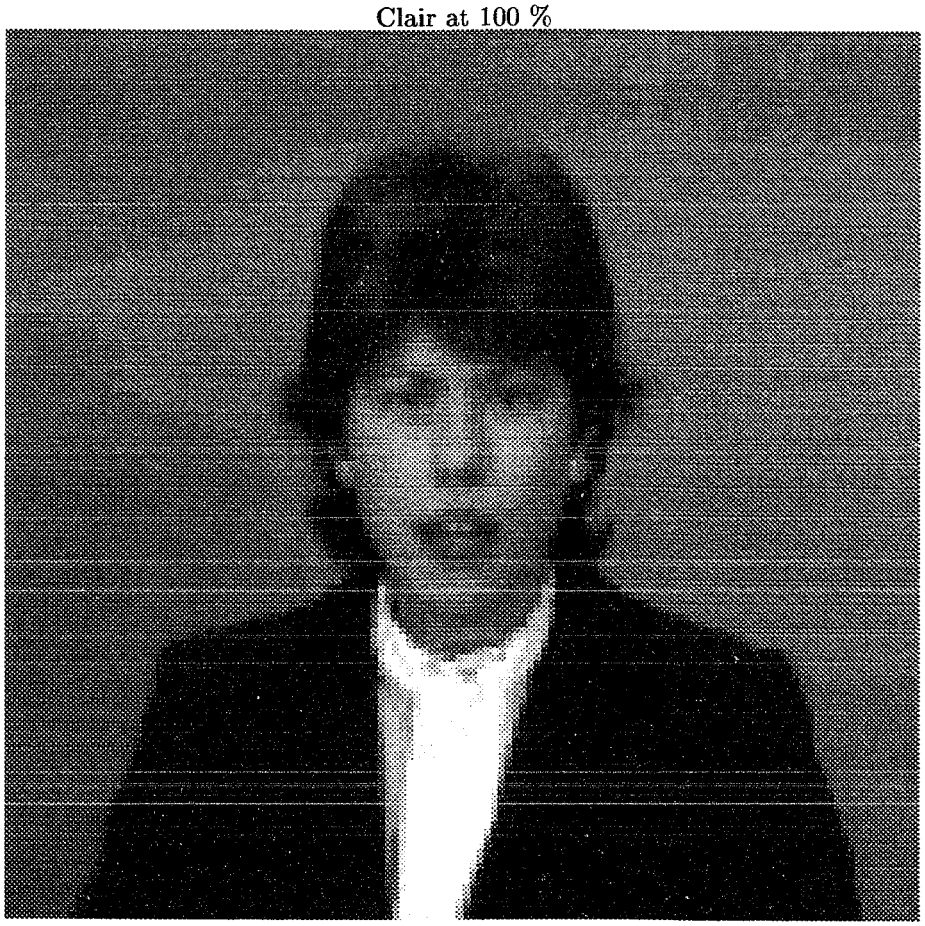


Figure 3.1a

Clair at 90 %

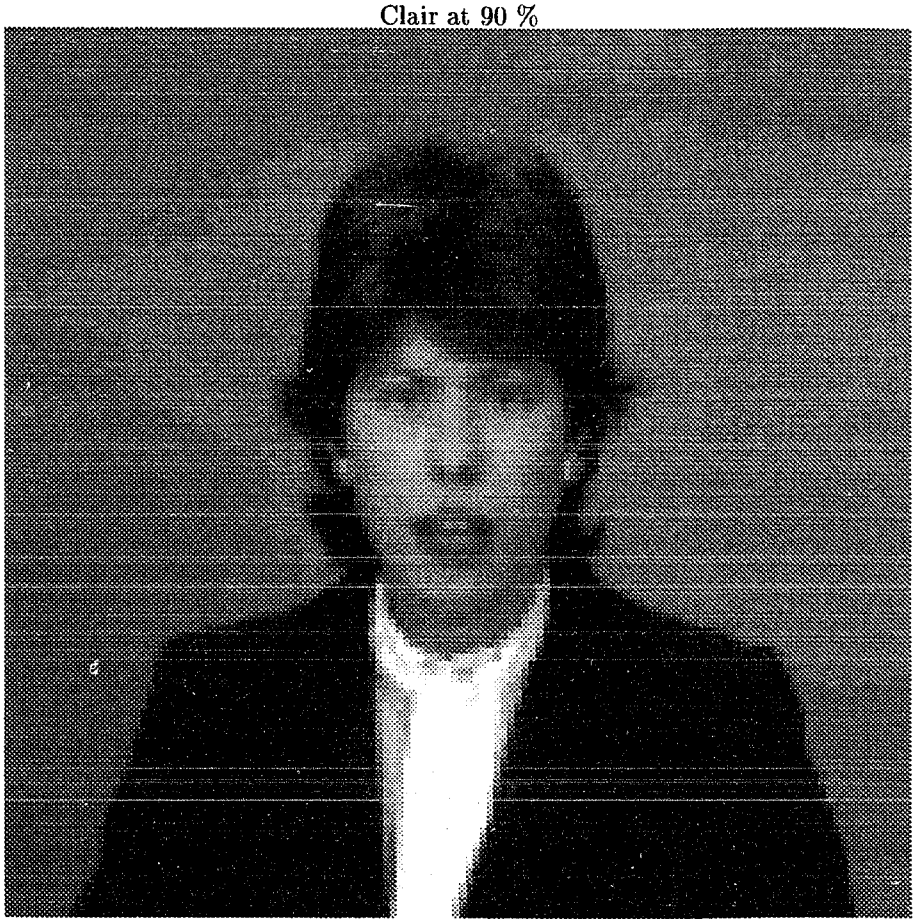


Figure 3.1b

Clair at 80 %

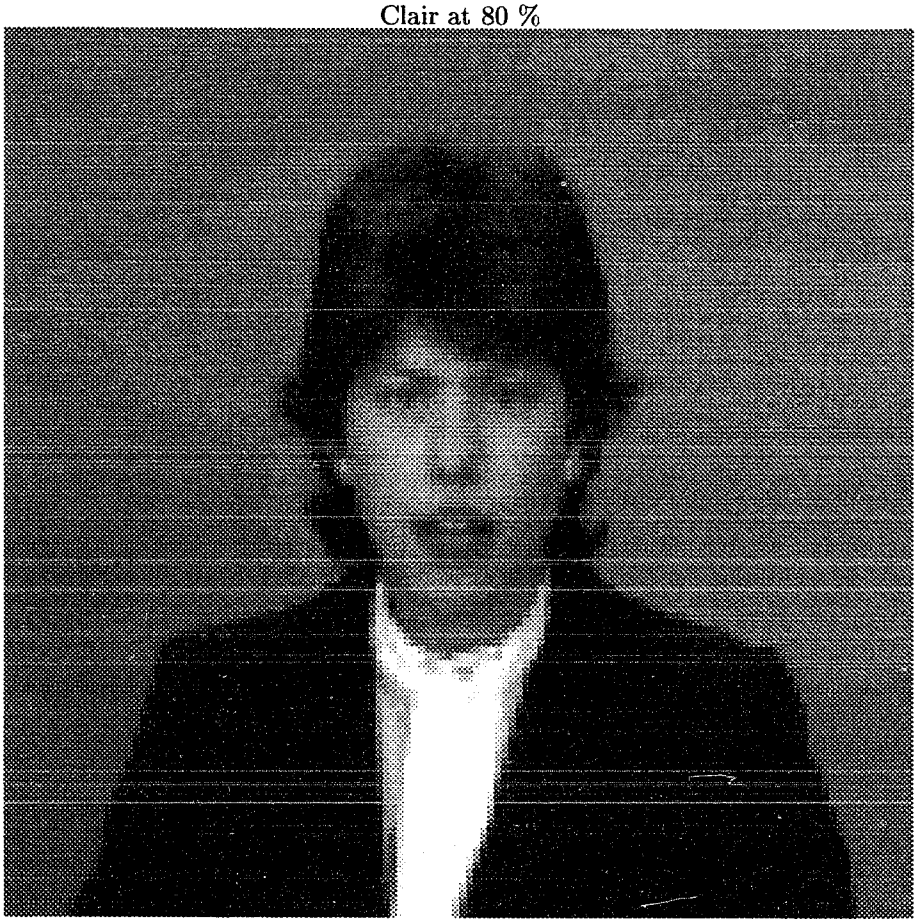


Figure 3.1c

Clair at 70 %

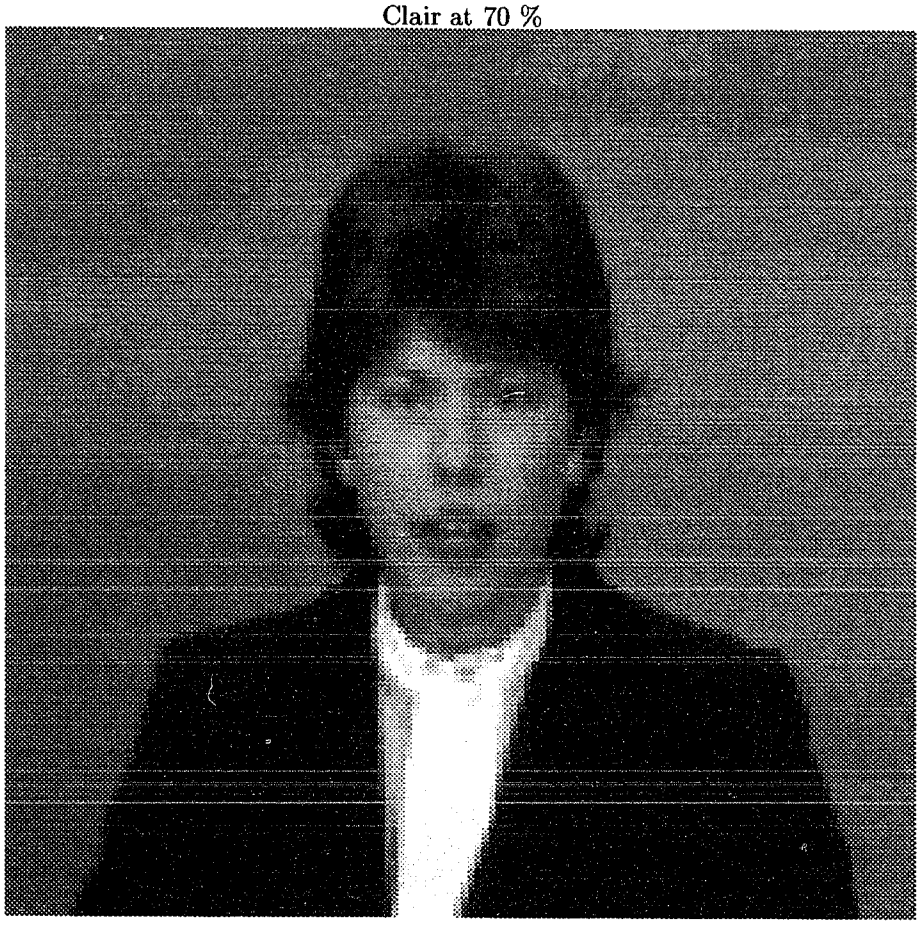


Figure 3.1d

Clair at 60 %

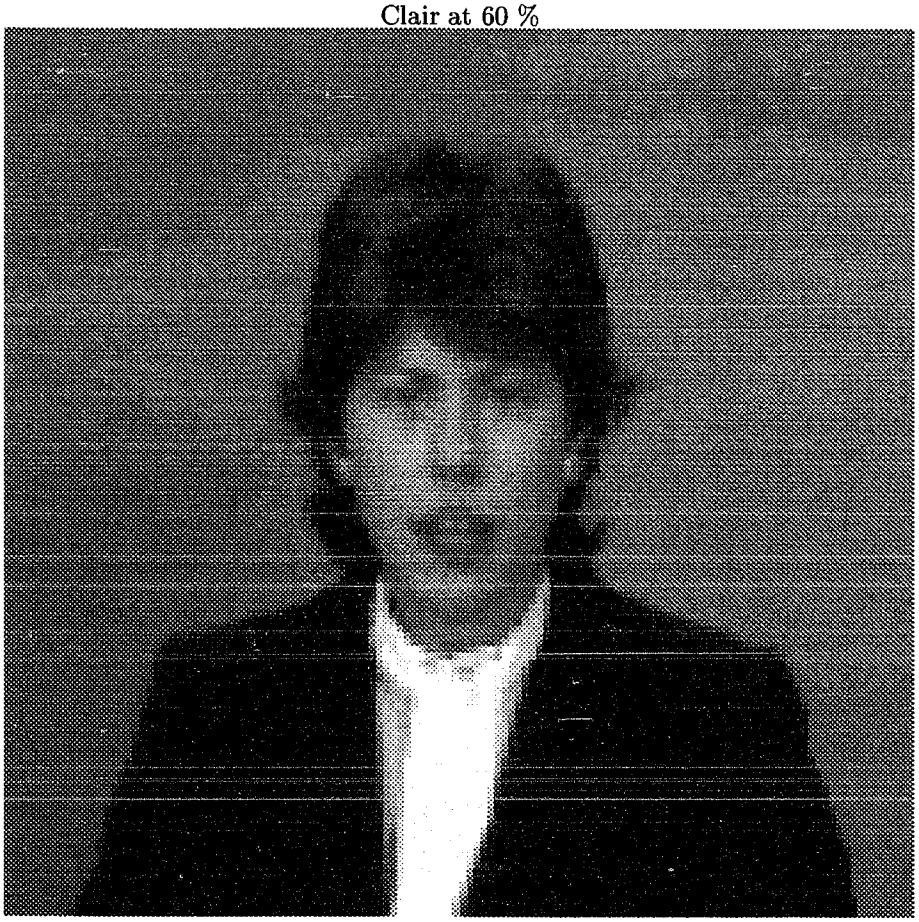


Figure 3.1e

Clair at 50 %

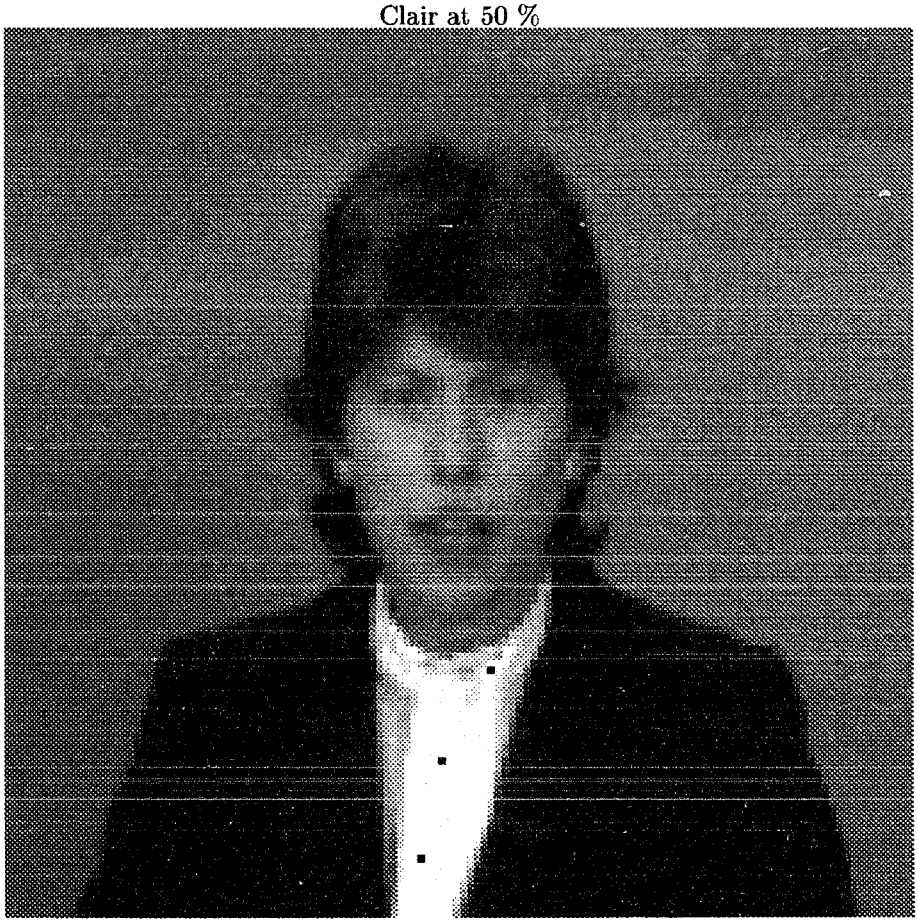


Figure 3.1f

Clair at 40 %

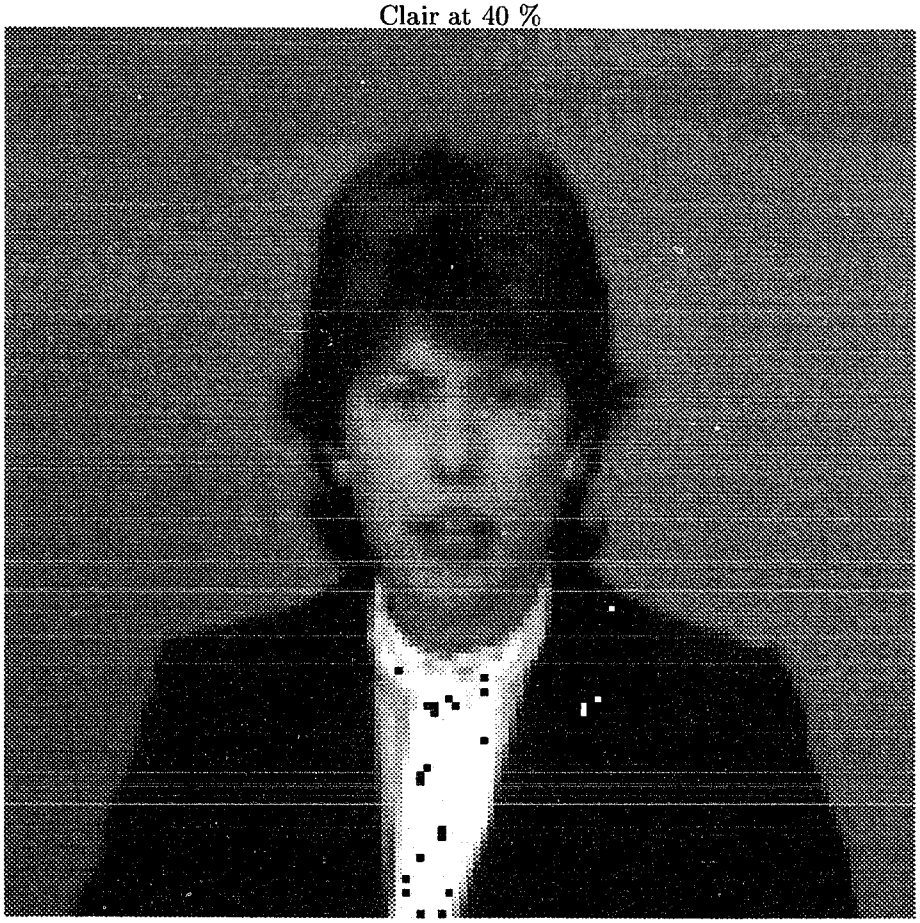


Figure 3.1g

Clair at 30 %

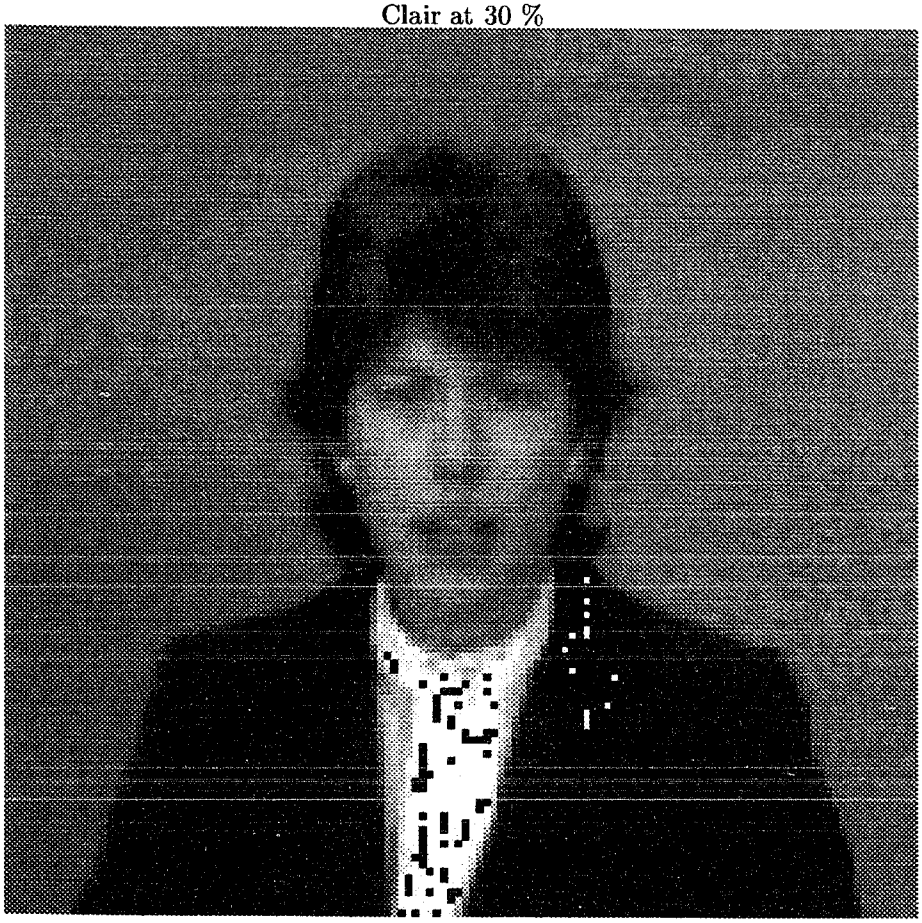


Figure 3.1h

Clair at 20 %

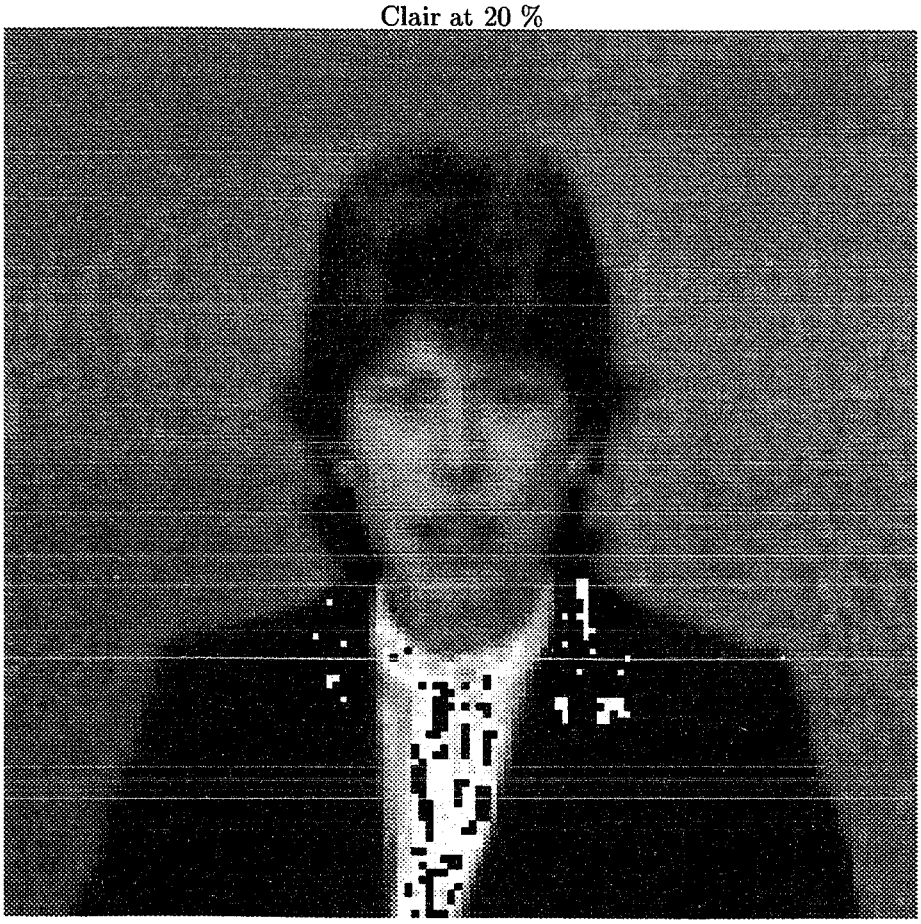


Figure 3.1i

Clair at 10 %

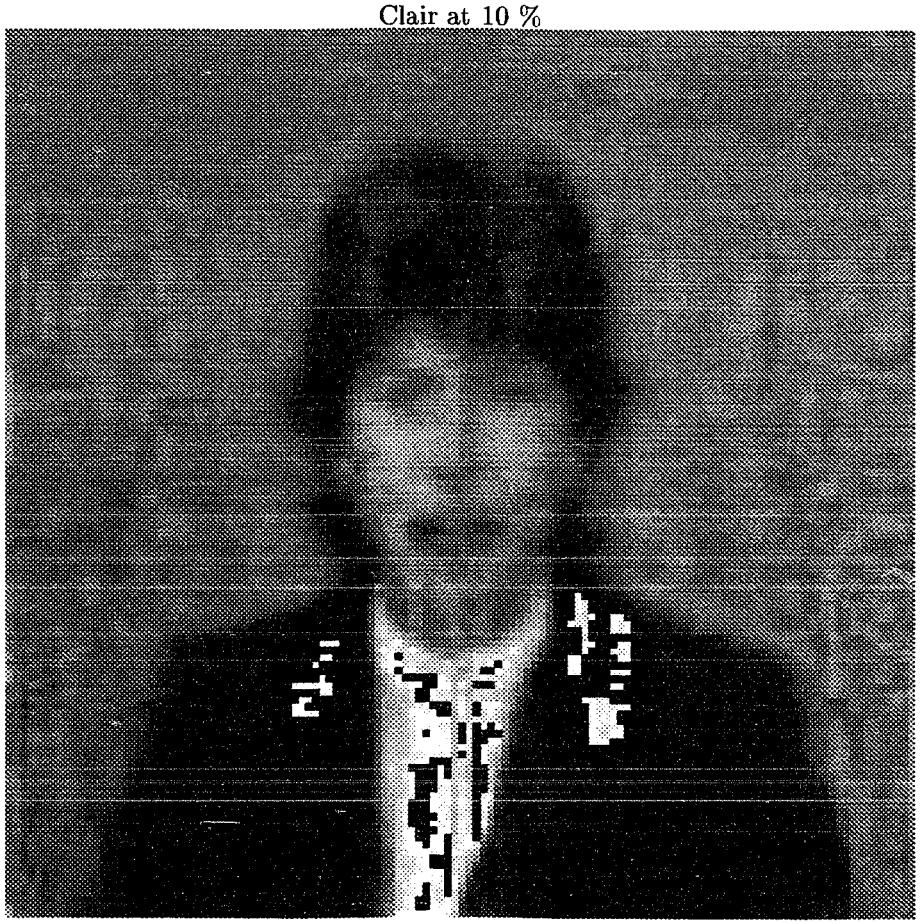


Figure 3.1j

Clair at 5 %

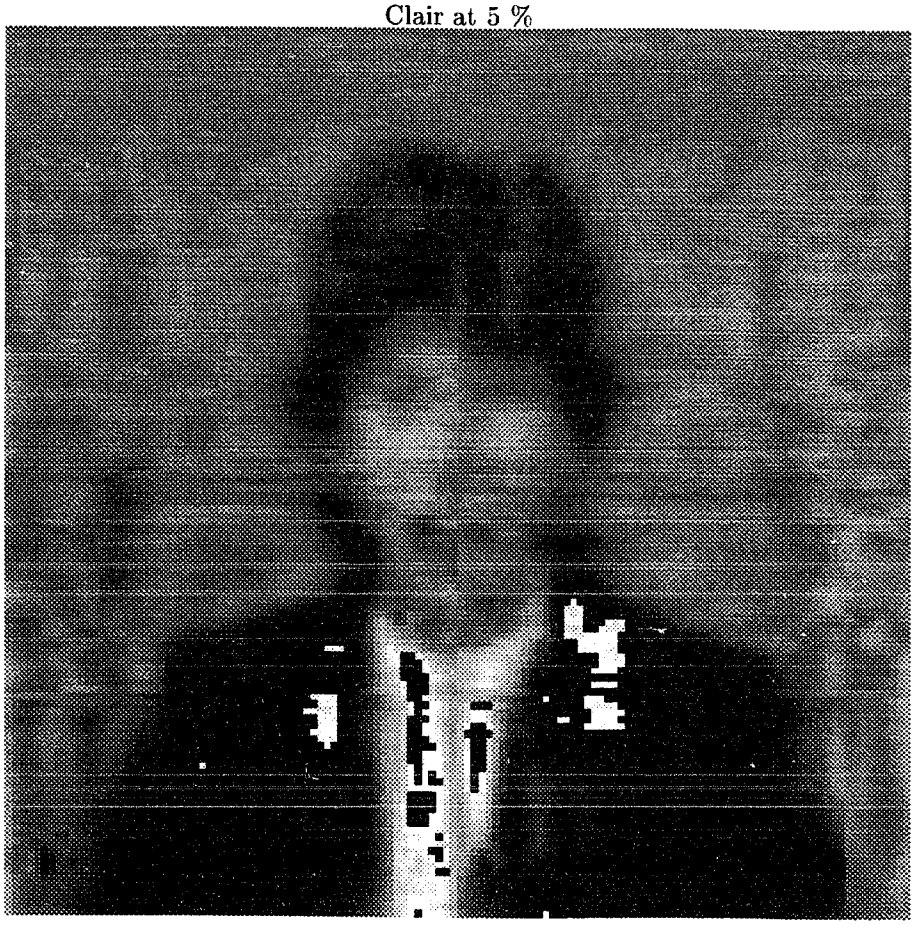


Figure 3.1k

Clair at 3 %

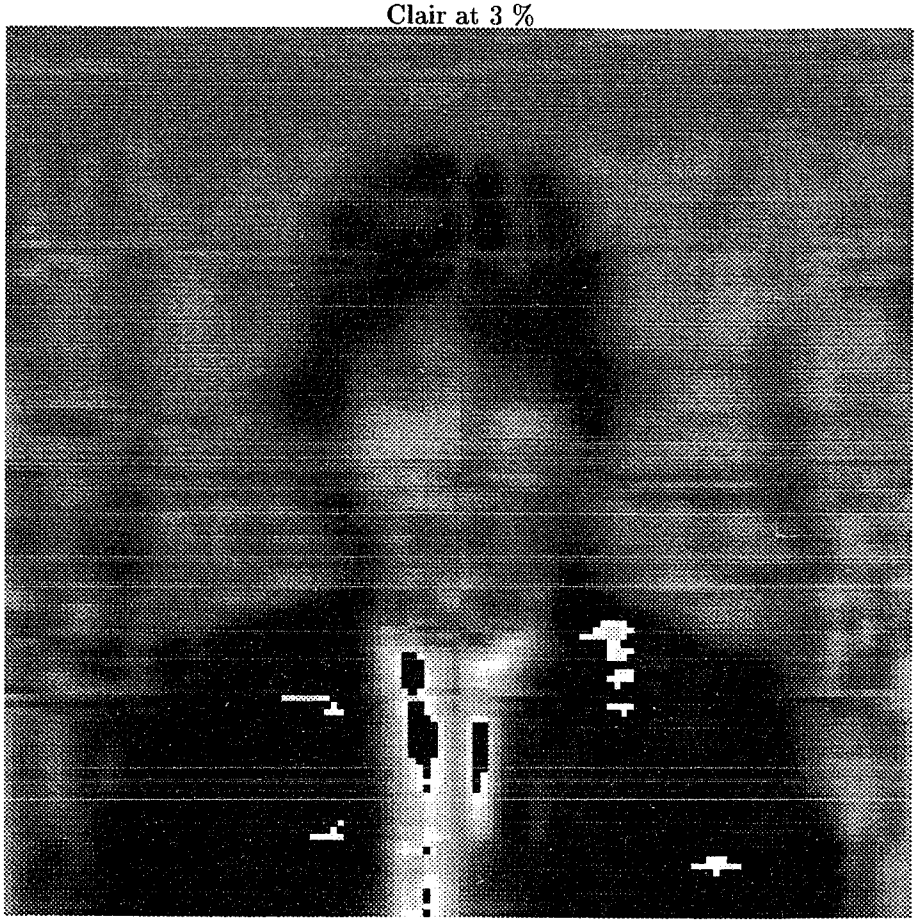


Figure 3.11

Clair at 2 %

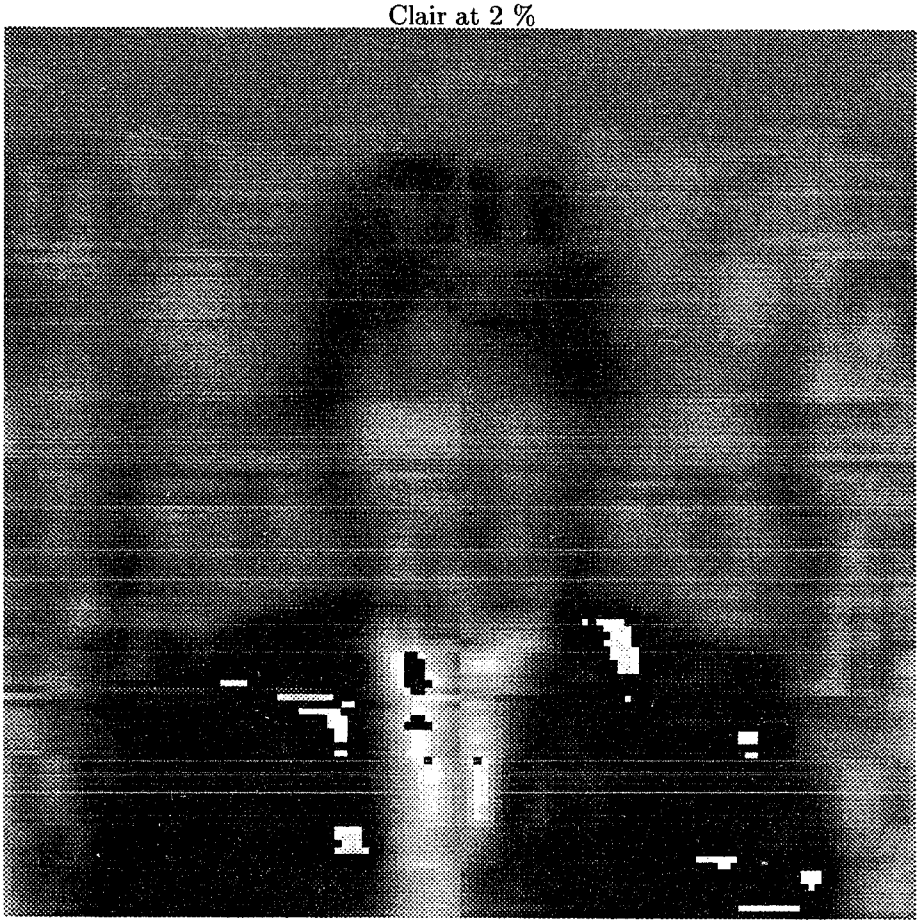


Figure 3.1m

Clair at 1 %

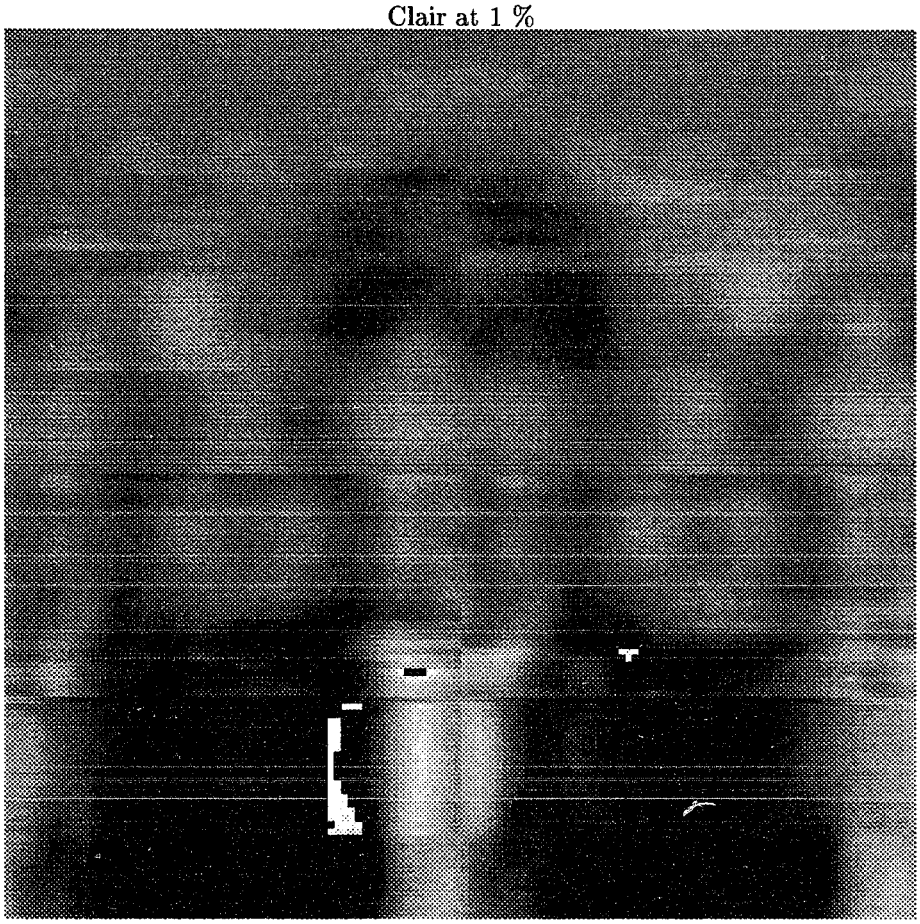


Figure 3.1n

3.4 Conclusions

We read numerous papers and books on the techniques of image processing and image compression. The purpose of this thesis was to process the available information, which contained a great deal of purely theoretical stuff,

and to write the application software on the top of it. We saw evidence that wavelet processing has great promise as a fast and convenient technique for image compression. Objective assessment of the relative merits of different processes is difficult, because the meaning of words "important details" are different in different application. Details that are essential for location of military targets in a landscape will be different from those that convey expression in a human face. In the same time from analysis of human face picture we believe that wavelets will have wide application in both military and civil fields. We didn't have the objective to do systematic comparison of the performance of wavelet and other processes applied to the same image with the same criteria of merit.

Bibliography

[1] H. Abarbanel, K. Case, F. Dyson, et al. *Wavelets.* The MITRE Corporation, McLean, Virginia, 1992.

[2] A. H. Akansu and R. A. Haddad. The binomial qmf-wavelet transform for multiresolution signal decomposition. *IEEE Transactions on Signal processing*, 1991.

[3] A. H. Akansu and Y. Lui. On signal decomposition techniques. *Optical Engineering*, 30(7):912–920, July 1991.

[4] A. N. Akansu and Y. Liu. Tree structures for time-frequency signal analysis. *Processing ICASSP-91*, 3:2037–2040, 1991. Toronto.

[5] L. Auslander, T. Kailath, and S. Mitter. *Signal processing*, volume 22, chapter Part I : Signal Processing Theory. Springler-Verlag, New - York, 1990.

[6] L. Auslander and R. Tolimeiri. *On finite Gabor expansion of signals*, volume 22, chapter Signal processing. Part I : Signal Processing Theory. Springler-Verlag, New - York, 1990.

[7] R. Balart. Matrix reformulation of the gabor transform. *Optical Engineering*, 31(6):1235–1242, June 1992.

[8] M. Basseville and A. Benveniste. Multiscale statistical signal processing. *Proceeding ICASSP-89*, pages 2065–2068, 1989. Glasgow.

[9] M. Basseville, A. Benveniste, and A. S. Willsky. Multi-scale autoregressive processes. *Publication Interne IRISA/INRIA*, (525), March 1990.

[10] M. J. Bastiaans. A sampling theorem for the complex spectrogram, and gabor's expansion of a signal in gaussian elementary signals. *Optical Engineering*, 20(4):594–598, July/August 1981.

[11] A. Benveniste, M. Basseville, R. Nikoukhah, A. S. Willsky, and K.C. Chou. Multiscale statistical signal processing and random fields on homogeneous trees. *Publication Interne IRISA/IRISA*, March 1991.

[12] G. Beylkin, R. Coifman, and V. Rokhlin. Fast wavelet transforms and numerical algorithms. *Pure and Applied math*, XLIV:141–183, 1991.

[13] T. Chen and P.P. Vaidyanathan. Multidimensional multirate filters and filter banks derived from one dimensional filters. *Electronics Letters*, pages 225–228, January 1991.

[14] K. C. Chou, S. Golden, and A. S. Willsky. Modeling and estimation of multiscale stochastic processes. *Proceeding ICASSP-91*, pages 1709–1712, May 1991.

[15] C. K. Chui. *Approximation Theory and Functional Analysis*, chapter An overview of wavelets, pages 47–71. Academic Press, Boston, 1991.

[16] Mac A. Cody. The fast wavelet transform. *Dr. Dobb's Journal*, pages 16–28, April 1992.

[17] A. Cohen. *Wavelets: A Tutorial in Theory and Applications*, pages 123–152. Academic Press, 1992.

[18] A. Cohen and Ingrid Daubechies. Orthonormal bases of compactly supported wavelets. better frequently resolution. *SIAM Journal of Mathematical Analysis*, 24(2):520–527, March 1993.

[19] A. Cohen, Ingrid Daubechies, and J.-C. Feauveau. Biorthogonal bases of compactly supported wavelets. *Commun. Pure and Applied Mathematics*, 45:485–560, 1992.

[20] Jack. K. Cohen. Battle-lemarie wavelets. *Colorado school of mines*, 1993.

[21] R. R. Coifman. *Wavelet analysis and signal processing*, volume 22, chapter Signal Processing, Part I: Signal Processing Theory. Springer-Verlag, New - York, 1990.

[22] J. M. Combes, A. Grossmann, and Ph. Tchamitchian. Wavelets: Time - frequency methods and phase space. In *Proceedings of the International Conference, Marseille, France*, New - York, 1989, December 1987. Springer - Verlag.

[23] R. E. Crochiere and L. R. Rabiner. *Multirate Digital Signal Processing.* Prentice - Hall, Englewood Cliffs, New Jersey, 1983.

[24] J. A. Crowe, N. M. Gibson, M. S. Woolfson, and M. G. Somekh. Wavelet transform as a potential tool for ecg analysis and compression. *Biomedical Engineering*, 14:268–272, May 1992.

[25] G. Cybenko and B. Usevitch. *Proceedings of 2nd International Workshop SVD and Signal Processings*, pages 197–208. Dept. of ELE, URI, Kingston, June 1990.

[26] Ingrid Daubechies. Time - frequency localization operators: a geometric phase space approach. *IEEE Transaction Information Theory*, 34(4):605–612, July 1988.

[27] Ingrid Daubechies. Orthonormal bases of compactly supported wavelets. *Pure and Applied Math*, 41:909–996, 1989.

[28] Ingrid Daubechies. The wavelet transform, time - frequency localization and signal analysis. *IEEE Transaction Information Theory*, 36(5):961–1005, September 1990.

[29] Ingrid Daubechies. Ten lectures on wavelets. In *CBMS-NSF Regional Conference Series in Applied Mathematics*, number 61, Philadelphia, PA, 1992. Society for Industrial and Applied Mathematics.

[30] Ingrid Daubechies. Orthonormal bases of compactly supported wavelets. variations on a theme. *SIAM Journal of Mathematical Analysis*, 24(2):499–519, March 1993.

[31] Ingrid Daubechies, Alex Grossmann, and Y. Meyer. Painless nonothogonal expansions. *Journal of mathematics and Physics*, 27(5):1271–1283, May 1986.

[32] N. G. De Bruijn. *Uncertainty principles in Fourier analysis*, pages 57–71. Academic Press, New York, 1967.

[33] P. A. DeVore, B. Jawerth, and B. J. Lucier. Image compression through wavelet transform coding. *IEEE Transaction Information Theory*, 38(2):719 - 746, March 1992. Theory: Special issue on Wavelet Transform and Multiresolution Signal Analysis.

[34] C. Dorize and L. F. Villemoes. Optimizing time-frequency resolution of orthogonal wavelets. *Proceesing ICASSP-91*, 1991.

[35] M. Doroslova, H. Fan, and P. M. Djuri. Discrete-time wavelets: time-frequency localization, shift-invariance, modeling of linear time-invariant systems. *Proceedings Conf. Inform. Sciences and Systems*, March 1992.

[36] G. Evangelista. Orthogonal wavelet transforms and filter banks. In *Twenty Third Asilomar Conf. Circuits, Systems and Computers*, volume 1, pages 489–492, Pacific Grove, 1989.

[37] G. Evangelista and C. W. Barnes. Discrete time wavelet transforms and their generalizations. *Proceedings ISCAS-90*, pages 2026–2029, 1990.

[38] Paul Michael Farrelle. *Recursive Block coding for Image Data Compression.* Springer-Verlag, New York, 1990.

[39] P. Flandrin. On the spectrum of fractional brownian motions. *IEEE Transaction Information Theory*, 35(1):197–199, January 1989.

[40] P. Flandrin, F. Magand, and M. Zakharia. Generalized target description and wavelet decomposition. *IEEE Transactions Acoustic, Speech and Signal Processing*, 38(2):350–352, February 1990.

[41] P. Franklin. A set of continuous orthogonal functions. *Mathematical Analysis*, (100):522–529, 1928.

[42] Michael W. Frazier and Arun Kumar. *An introduction to the orthonormal wavelet transform on discrete set*, chapter 2. CRC Press, Inc., 1994.

[43] Michael W.B. Frazier and John Benedetto, editors. *Wavelets: mathematics and application.* CRC Press, 1994.

[44] Michael W.B. Frazier, Jawerth, and G. Weiss. Littlewood-paley theory and the study of function spaces. In *CBMS Regional Conference*

Series in Mathematics, number 79, Providence, RI, 1991. American mathematical Society.

[45] M. Frisch and H. Messer. Detection of a transient signal of unknown scaling and arrival time using the discrete wavelet transform. *Proceedings ICAASSP-91*, pages 1313–1316, May 1991.

[46] L. Gagnon and J.M. Lina. Wavelets and numerical split-step method: A global adaptive scheme, June 1994. This work is supported through funds provided by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

[47] D. Garreau. Multiscale inverse filtering. *Proceedings ICAASSP-90*, pages 2495–2498, 1990.

[48] R. A. Gopinath and C. S. Burrus. A tutorial overeview of filter banks, wavelets and interrelations. *Proceedings ISCAS-93*, pages 1–4, 1993.

[49] R. A. Gopinath, W. M. Lawton, and C. S. Burrus. Wavelet-galerkin approximation of liner translation invariant operators. *Proceedings ICAASSP-91*, 1991.

[50] A. Grossmann and J. Morlet. Decomposition of hardy functions into square integrable wavelets of constant shape. *SIAM J. Math. Anal*, 15(4):723–736, July 1984.

[51] A. Grossmann and J. Morlet. Decomposition of functions into wavelets of constant shape and related transforms. *Mathematics and Physics, lectures and Recent Results*, 1985.

[52] A. Grossmann, J. Morlet, and T. Paul. Transforms associated to square integrable group representations. general results. *Mathematical Physics*, (26):2473–2479, 1985.

[53] C. E. Heil and D. F. Walnut. Continuous and discrete wavelet transforms. *SIAM Review*, 31(4):628–666, December 1989.

[54] Christopher Heil. Methods of solving dilation equation. *Mathematics Department. MIT*, 1994.

[55] Loren Heiny. *Advanced graphics programming using C/C++*. John Wiley and Sons, Inc., 1993.

[56] C.W. Helstrom. An expansion of a signal in gaussian elementary signals. *IEEE Transaction Information Theory*, 12(1):628–666, 1966.

[57] C. Herley and M. Vetterli. Linear phase wavelets: Theory and design. *Proceedings ICAASSP-91*, 1991.

[58] M. Holschneider. Wavelet analysis on the circle. *Mathematical Physics*, 31(1):39–44, January 1990.

[59] E. M. Janssen. Bargmann transform, zak transform, coherent states. *Mathematical Physics*, 23(5):720–731, May 1982.

[60] E. M. Janssen. Gabor representation and wigner distribution of signals. *Proceedings ICASSP-84*, 1984.

[61] Bjorn Jawerth, Ronald A. DeVore, and Bradley J. Lucier. Image compression through wavelet transform coding. *IEEE Transactions on Information Theory*, 38(2), March 1992.

[62] Bjorn Jawerth and Wim Sweldens. An overview of wavelet based multiresolution analysis. *National Fund of Scientific Research Belgium*, 1993.

[63] H. E. Jensen and J. Justensen. Double series representation of bounded signals. *IEEE Transaction Information Theory*, 34(4):613–624, July 1988.

[64] S. Kadambe and G. F. Boudreaux-Bartels. A comparison of a wavelet functions for pitch detection of speech signals. *Proceedings ICASSP-91*, pages 449–452, 1991.

[65] Gunnar Karlsson and Martin Vetterly. Extension of finite length signals for sub-band coding. In *Signal Processing*. Elsevier Science Publishers B. V., October 1988.

[66] G. Knowles. Vlsi architecture for the discrete wavelet transform. *Electronic Letters*, 26(15):1184–1185, July 1990.

[67] J. Kovacevic and M. Vetterli. Perfect reconstruction filter banks with rational sampling rate changes. *Proceedings of ICASSP*, 3:1785–1788, May 1991.

[68] A. Kumar and D. R. Fuhrmann. The frazier-jawerth transform. *Proceedings ICASSP-90*, pages 2483– 2486, 1990.

[69] W. Lawton. Tight frames of compactly supported affine wavelets. *Mathematical physics*, 31:1898–1901, 1990.

[70] W. Lawton. Necessary and sufficient conditions for constructing orthonormal wavelet bases. *Mathematical physics*, 32:57–61, 1991.

[71] P. G. Lemarie. *Lecture notes in Mathematics*, volume 1438, pages 1–13. Springer-Verlag, New York, 1990.

[72] P. G. Lemarie. *Lecture notes in Mathematics*, volume 1438, pages 26–38. Springer-Verlag, New York, 1990.

[73] A. S. Lewis and G. Knowles. Video compression using 3d wavelet transform. *Electronic Letters*, 26(6):396–398, March 1990.

[74] Jae Lim and S.Naveed A. Malik. A new algorithm for two-dimensional maximum entropy power spectrum estimation. *IEEE Transaction on Acoustic, Speech, and Signal Processing*, ASSP-29(3), June 1981.

[75] Jean-Marc Lina and Michel Mayrand. Parametrizations for daubechies wavelets. *Rapid Communications*, 48(6):48–51, December 1993.

[76] Stephane Mallat and Sifen Zhong. Characterization of signals from multiscale edges. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 14:710–752, July 1992.

[77] Stephane G. Mallat. Multifrequency channel decompositions of images and wavelet models. *IEEE Transactions Acoustic, Speech and Signal Processing*, 37(12):2091–2110, December 1989.

[78] Stephane G. Mallat. Multiresolution approximations and wavelet orthonormal bases of $l^2\mathbf{R}$. *Transaction of American Mathematical Society*, 315(1):69–87, September 1989.

[79] Stephane G. Mallat. A theory for multiresolution signal decomposition: The wavelet representation. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 11(7):674–693, July 1989.

[80] Stephane G. Mallat. Zero-crossings of a wavelet transform. *IEEE Transaction Information Theory*, 37(4):1019–1033, July 1991.

[81] S. Mann and S. Haykin. An adaptive generalization of the wavelet transform. *Optical Engineering*, 31(6):1243–1256, June 1992.

[82] Bruce F. Mcguffin and Bede Liu. An efficient algorithm for two-dimensional autogressive spectrum estimation. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(1):106–117, January 1989.

[83] Charles A. Micchelli. Using the refinement equation for construction of pre-wavelets. Technical report, Department of Mathematical Sciences, IBM, 1990.

[84] J. Morlet, G. Arens, E. Fourgeau, and D. Giard. Wave propagation and sampling theory. *Geophysics*, 47(2):203–236, February 1982.

[85] Shgeru Muraki. Approximation and rendering of volume data using wavelet transforms. *IEEE Computer Graphics and Applications*, pages 21–28, 1992.

[86] Shgeru Muraki. Volume data and wavelet transforms. *IEEE Computer Graphics and Applications*, pages 50–56, 1993.

[87] John R.Bruce O'Hair and W. Suter. Eliminating distortion in the beylkin-coifman-rokhlin transform. Wright-Patterson AFB, OH 45433. Department of Electrical and Computer Engineering, Air Force Institute of Technology.

[88] Dick Oliver, Scott Anderson, James McCord, Spyro Gumas, and Bob Zigon. *Tricks of graphics gurus.* SAMS Publishing, Carmel, IN, 1993.

[89] T. W. Parks and R. G. Shenoy. Time-frequency concentrated basis functions. *Proceedings ICAAASSP-90*, pages 2459–2462, 1990.

[90] B. William Pennebaker. *JPEG Still Image Data Compression Standard.* Van Nostrand Reinhold, New York, 1993.

[91] Ashok Popat, Wei Li, and Murat Kunt. Numerical design of parallel multiresolution filter banks for image coding applications. Technical report, Signal Processing Laboratory, Swiss Federal Institute of Technology, 1015 Lausanne, Switzerland, 1991.

[92] William H.Saul Press, A. Teukolsky, William T. Vetterling, and Brian P.Flannery. *Numerical Recipes in C.* Cambridge University Press, New York, 1992.

[93] J. Ramanathan and O. Zeitouni. On the wavelet transform of fractional brownian motion. *IEEE Transaction Information Theory*, 37(4):1156–1158, July 1991.

[94] P. A. Regalia, S. K. Mitra, and P. P. Vaidyanathan. The digital allpass filter: a versatile signal processing building block. *Proceeding IEEE*, 76(1):19–37, January 1988.

[95] Chistopher E. Reid and Thomas B. Passin. *Signal Processing in C.* John Wiley and Sons, Inc., 1992.

[96] H. L. Resnikoff. Wavelets and adaptive signal processing. *Optical Engineering*, 31(6):1229–1234, June 1992.

[97] S. D. Riemenscheider and Z. Shen. *Approximation Theory and Functional Analysis*, chapter Box Splines, Cardinal Series, and Wavelets, pages 133–149. Academic Press, Boston, 1991.

[98] O. Rioul and M. Vetterli. Wavelets and signal processing. *IEEE Signal Processing Magazine*, 8(4):14–38, October 1991.

[99] O. Roiul. A discrete-time multiresolution theory. *IEEE Transaction Signal Processing*, 1991. preprint.

[100] M. B. Ruskai. *Wavelets and Their Applications.* Jones and Bartlett, Boston, MA, 1992.

[101] M.T.J. Smith and S.L. Eddins. Sub-band coding of images with octave band tree structures. *IEEE Transactions Acoustic, Speech and Signal Processing*, pages 1382–1385, April 1987.

[102] Gilbert Strang. Wavelets and dilation equations: a brief introduction. *SIAM Review*, 31(4):614–627, December 1989.

[103] Gilbert Strang. Wavelet transforms versus fourier transform. *Bulletin (New Series) of the American Mathematical Society*, 28(2):288–305, April 1993.

[104] Robert S. Strichartz. *An introduction to the orthonormal wavelet transform on discrete set*, chapter 1. CRC Press, Inc., 1994.

[105] J.O. Stromberg. A modified franklin system and higher order spline systems on $\mathbf{R}^n$ as unconditional bases for hardy spaces. *Conference on Harmonic Analysis in Honor of Antoni Zygmund*, 2:475–494, 1981.

[106] James R. Sullivan, Majid Rabbani, and Benjamin M. Dawson, editors. *Image Processing Algorithms and Techniques III*, volume 1657, San Jose, California, February 1992. The Society for Imaging Science and Technology.

[107] Carl Taswell and Kevin C. McGill. Wavelet transform algorithms for finite-duration discrete-tome signals. Technical report, Department of Computer Science, Stanford University, October 1993.

[108] B. Torresani. Wavelets associated with representations of the affine weyl-heisenberg group. *Mathematical physics*, 32(5):1273–1279, 1991.

[109] P. P. Vaidyanathan. Multirate digital filters, filter banks, polyphase network, and applications: a tutorial. *Proceedings IEEE*, 78(1):56–93, January 1990.

[110] P. P. Vaidyanathan and V. C. Liu. Classical sampling theorems in the context of multirate and polyphase digital filter bank structures. *IEEE Transactions Acoustic, Speech and Signal Processing*, 36(9), September 1988.

[111] M. Vetterli and C. Herley. Wavelets and filter banks: relationships and new results. *Proceedings ICASSP-90*, pages 1723–1726, 1990.

[112] M. Vetterly and C. Herley. Wavelets and filter banks: theory and design. *IEEE, Trans. in ASSP*, pages 2207–2232, September 1992.

[113] G. W. Wornell. A karhunen-lo expansion for $1/f$ processes via wavelets. *IEEE Transaction Information Theory*, 36(4):859–861, July 1990.

[114] G. W. Wornell and A. V. Oppenheim. Estimation of fractal signals from noise measurements using wavelets. *IEEE Transaction on Signal Processing*, 40(3):611–623, March 1992.

[115] Q. Zhang and A. Benveniste. Approximation by nonlinear wavelets networks. *Proceedings ICASSP-91*, pages 3417–3420, 1991.